An Implementation Substrate for Languages Composing Modularized Crosscutting Concerns

Hans Schippers[*]
University of Antwerp
Antwerp, Belgium
hans.schippers@ua.ac.be

Michael Haupt
Hasso-Plattner-Institut
Uni. Potsdam, Germany
michael.haupt@hpi.uni-potsdam.de

Robert Hirschfeld
Hasso-Plattner-Institut
Uni. Potsdam, Germany
hirschfeld@hpi.uni-potsdam.de

ABSTRACT

We present the implementation of several programming languages with support for multi-dimensional separation of concerns (MDSOC) on top of a common delegation-based substrate, which is a prototype for a dedicated MDSOC virtual machine. The supported MDSOC language constructs range from aspects, pointcuts and advice to dynamically scoped and activated layers. The presented language implementations show that the abstractions offered by the substrate are a viable target for high-level language compilers.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*run-time environments*

Keywords

Crosscutting concerns, modularization, aspect-oriented programming, context-oriented programming, machine model, language implementation

1. INTRODUCTION

Programming language abstractions for crosscutting concern modularization and composition constitute a programming paradigm in its own right, which has been referred to as *multi-dimensional separation of concerns* (MDSOC) [19]. It has spawned several approaches, including aspect-oriented programming (AOP) [8] and context-oriented programming (COP) [7]. MDSOC language implementations, however, do not provide sufficient dedicated run-time environment support for MDSOC mechanisms.

In [6], we presented a machine model centered on the basic notions of objects, messages and delegation [9], which was showed to naturally support an important set of language mechanisms designed to modularize crosscutting concerns.

Subsequently, [4] provided mappings from the semantics of four languages supporting different modularization mechanisms to this model. More specifically, the *j* language family [17] was used. The *j* language itself is a Java subset that is gradually extended with different mechanisms for the modularization of crosscutting concerns. Here, *ij* adds inter-type declarations, while *aj* adds pointcuts and advice. *cj* [4] is an addition to the *j* language family. *cj* does not include the features of *ij* and *aj*, but instead adopts context-oriented programming (COP) [7], a layer-based approach to the modularization of crosscutting concerns. Layers allow context-specific behavioral variations to be composed based on the execution context.

In summary, the aforementioned semantic mappings for the *j* language family formally suggest a translation process from different MDSOC approaches to the basic functionality offered by our delegation-based machine model.

This paper presents an implementation of our machine model as an instruction set for a virtual machine (VM) with inherent MDSOC support. Moreover, it contributes the technical realization of the translation process, making the *j* language family executable on top of this delegation-based VM prototype. In particular, the presented language implementations are *j*, *cj* and *iaj*, the latter containing the features of both *ij* and *aj*. We emphasize once more that the implementations shown in this paper represent a proof-of-concept effort, intended to illustrate our machine model's viability to serve as a common substrate for MDSOC languages. Efforts to achieve performance comparable to existing OOP implementations belong to future work.

The paper is organized as follows. In the next section, we briefly revisit the machine model, and provide a short description of the framework that was used for its implementation. Sec. 3 describes the high-level *j*, *iaj* and *cj* programming languages, while Sec. 4 details their translation process to our instruction set constituting the implementation kernel, a presentation of which is also included in that section. Sec. 5 attends to related work, before the paper is summarized and future work is discussed in Sec. 6.

2. MACHINE-LEVEL SUPPORT FOR CROSSCUTTING CONCERNS

This section gives brief overviews of the execution model supporting the implementation substrate presented in this

[*]Ph.D. fellowship of the Research Foundation, Flanders (FWO)

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SAC'09 March 8-12, 2009, Honolulu, Hawaii, U.S.A.
Copyright 2009 ACM 978-1-60558-166-8/09/03 ...$5.00.

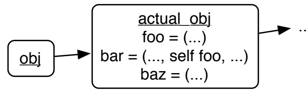

Figure 1: An object and its proxy.

paper, and of the implementation framework used to realize the substrate in software.

2.1 A Machine Model

The machine model we use in this work to implement several programming language approaches to modularizing crosscutting concerns has been introduced in [6].

Core features of the model pertain to the representation of application entities, and that of join points. The latter are regarded as *loci of late binding*, and hence of virtual functionality dispatch, where dispatch is organized along multiple dimensions. Each dimension is one possible way to choose a particular binding of a piece of functionality to a join point, e. g., the current object, the target of a method call, the invoked method, the current thread, etc.

Objects are, using a prototype-based object-oriented environment, represented as "seas of fragments" [12]: each object is visible to others only in the form of a *proxy.* Messages sent to an object are received by its proxy and *delegated* to the actual object, as displayed in Fig. 1. Classes are represented likewise: each class is a pair of a proxy and an object representing the actual class. Each object references its class by delegating to the class's proxy.

The granularity of the supported join point model is that of message receptions. Member field access is also mapped to messages. A join point's nature as a locus of late binding is realized by inserting additional proxy objects in between the proxy and the actual object, or in between the class object's proxy and the actual class-representing object. That way, a message passed on along the delegation chain can be interpreted differently by various proxies understanding it, establishing late binding of said message to functionality.

Weaving—both static and dynamic—is realized by allowing for the insertion and removal of proxy objects into and from delegation chains. The model is able to represent control-flow dependent advice and dynamic introductions in very natural and simple ways [6].

2.2 An Implementation Framework

For an implementation of both the model and programming languages on top of it, a framework supporting prototype-based object-oriented programming with Lieberman-style prototypes [9] is required. Lieberman-style prototypes ensure maximum flexibility by allowing an object to transparently delegate messages to any other object, whether or not it was originally cloned from the same parent prototype.

Another requirement for the implementation framework is that it should bring reflective capabilities strong enough to facilitate the dynamic modification of delegation chains between objects. Finally, it should support the implementation of programming languages.

The COLA (Combined Object-Lambda Abstractions) [13] project combines the aforementioned requirements in an elegant way, rendering it a useful framework for the purpose of implementing the model outlined in Sec. 2.1. We will now give a brief overview of it as of the time of this writing.

```
1  PROG    ::= CLS*
2  CLS     ::= class CLSNAME ext CLSNAME { DECL* }
3  DECL    ::= FIELD | METH
4  FIELD   ::= TYPE IDENT
5  METH    ::= TYPE IDENT ( TYPE x ) { EXPLST }
6  EXPLST  ::= EXP < ; EXP >*
7  EXP     ::= SPECIAL | VAR | new CLSNAME
8            | EXP . IDENT EXP . IDENT := EXP
9            | EXP . IDENT ( EXP ) | EXP == EXP
10           | if ( EXP ) { EXP } else { EXP }
11 SPECIAL ::= true | false | null | NUM | STR
12 VAR     ::= this | x
```

Listing 1: *j* syntax definition.

COLA consists of two main parts. The *object part* provides object-oriented abstraction by means of a minimal prototype-based object model [14] and a language with Smalltalk-like syntax that makes programming using the model possible. Both model and language are fully reflective and can bootstrap themselves.

On top of the object part, the *function part* provides functional abstraction. At this level, a programming language is available that represents C abstract syntax trees (ASTs) as S-expressions. The language also features macro definition and expansion capabilities as found in Scheme, making it a powerful tool for program transformation and generation. From within programs written for the function part of COLA, the underlying object part is still fully accessible. It is possible to choose between applicative and message-passing semantics. The function part of COLA is implemented using a just-in-time (JIT) compiler (realized as a COLA program itself) generating optimized native code.

An implementation of *parsing expression grammars* (PEGs) [3] was realized in the function part of COLA. With the PEG framework, it is very easy to define language implementations. Essentially, a rule of a PEG consists of grammar parts on the one hand, and of action parts on the other. Exploiting the features of COLA's function part—especially their macro-expansion capabilities—, it is straightforward to use the action parts of PEG rules to return COLA ASTs.

3. THE *j* LANGUAGE FAMILY

Next, we describe a number of minimalistic yet high level programming languages. All languages are based on *j* [17], a Java subset, and gradually add mechanisms to facilitate the implementation of crosscutting concerns.

3.1 The Core Language: *j*

j [17] is the core language and a subset of Java. Apart from a few very minor tweaks, such as the omission of *interfaces* (which are irrelevant since we do not focus on the static type system), the version presented in this paper is exactly the same. Its syntax is shown in Lst. 1.

A program consists of a number of classes, each inheriting from exactly one superclass and containing fields and methods. A method always accepts exactly one formal parameter which must be called `x`, and method bodies are composed of a list of expressions.

Local variables are not included, as they can be easily simulated by assigning values to fields of the formal parameter. Similarly, multiple parameters can be passed by encapsulating them in a single object, and passing that object instead.

```
1  PROG     ::= < CLS | ASP >*
2  CLS      ::= class CLSNAME ext CLSNAME { DECL* INTR* }
3  ASP      ::= aspect CLSNAME { DECL* INTR* ADVICE* }
4  INTR     ::= CLSNAME <-- DECL
5  ADVICE   ::= < before | after | TYPE around >
6               PNTCT { EXPLST }
7  PNTCT    ::= BPNTCT
8               < && cflow ( TYPE TYPE.IDENT(TYPE) ) >?
9  BPNTCT   ::= call ( TYPE TYPE.IDENT(TYPE) )
10            | get ( TYPE TYPE.IDENT )
11            | set ( TYPE TYPE.IDENT )
12 EXP      ::= ... | proceed ( EXP? )
13 VAR      ::= this | x | s | r | v
```

Listing 2: Differences between *j* and *iaj* syntax.

```
1  PROG     ::= < CLS | LAYER >*
2  CLS      ::= class CLSNAME ext CLSNAME { EXTDECL* }
3  EXTDECL  ::= DECL | CLAYER
4  LAYER    ::= layer CLSNAME { LDECL* }
5  LDECL    ::= FIELD | LMETH
6  LMETH    ::= TYPE CLSNAME.IDENT ( TYPE x ) { EXPLST }
7  CLAYER   ::= layer CLSNAME { DECL* }
8  EXP      ::= ... | withlayer ( CLSNAME ) { EXP }
9             | withoutlayer ( CLSNAME ) { EXP }
10            | proceed ( EXP )
11 VAR      ::= thisLayer | this | x | s
```

Listing 3: Differences between *j* and *cj* syntax.

Constants such as strings, numbers, `true`, `false` and `null` are primitive expressions, as are occurrences of `this` and the formal parameter `x`. More complex expressions can be formed in a very limited number of ways. In order of presentation, only class instantiation (via `new`), field access (both get and set), method invocation and comparison for equality are allowed, as well as an if-then-else construct. Finally, `IDENT`, `CLSNAME` and `TYPE` are all considered to be identifiers.

3.2 Introductions, Pointcuts and Advice: *iaj*

ij and *aj* were originally introduced as two separate extensions to *j* in [17], with *ij* adding inter-type declarations, and *aj* adding aspects with pointcuts and (before and after) advice. We present one combined language *iaj* here. Also, unlike the original *aj*, the version presented here supports *around* advice and *cflow* pointcuts. Hence, *iaj* can be regarded as a subset of AspectJ. The syntactical differences to *j* are listed in Lst. 2.

The definition of a class is extended to include inter-type declarations, which are described by the `INTR` rule. Each inter-type declaration adds either a field or a method to the definition of another (previously defined) class.

Apart from classes, a program may now include any number of *aspects*, which can contain *advice*. Advice may be subject to a cflow specification, meaning it should only apply provided the join point in question occurs within the control flow of a particular method. Supported join points include method calls and field access.

In advice bodies, some extra predefined variables are made available. More specifically `s`, denoting the caller object, `r`, the intended receiver object, and `v`, the value which should be assigned to a field in case of a `set` pointcut.

3.3 Context-oriented Programming: *cj*

Unlike *iaj*, the *cj* extension to *j* implements the concepts of *context-oriented programming* (COP) [7].

Context-oriented programming helps developers to modularize context-dependent behavior. Behavioral variations, or partial definitions of the underlying programming system, are organized in layers where each layer aggregates a context-dependent part of a system's property or concern. Layers can be activated and deactivated at runtime.

cj is implemented as an extension to *j*. The syntactic differences are shown in Lst. 3. Layers can be defined at top level as well as class level. A top-level layer definition contains method definitions that directly pertain to certain classes as expressed in the `LMETH` rule. Class-level layers implicitly affect methods of the surrounding class, hence they rely on the (unmodified) `METH` rule. The `EXP` rule has been extended with expressions for activating and deactivating a given layer, and for proceeding execution in the next layer or invoking the original method implementation.

While `s` has similar meaning as with *iaj*, an additional `thisLayer` keyword is needed. In a `withlayer` block, it can be used to access layer fields, while `this` is used to access fields belonging to the class the code occurs in.

4. IMPLEMENTING THE *j* LANGUAGES

In this section, we will first present a kernel implementing the core mechanisms of the machine model for MDSOC languages. The kernel consists of the basic implementation of the delegation-based machine model, as well as a number of core abstractions heavily relied on during the translation process. We will then present implementations of each of the languages described in Sec. 3.

4.1 The Kernel

As stated in Sec. 2.1, the representation of objects as "seas of fragments" is a core feature of the kernel. Consequently, the starting point of the kernel is `ProxifiedObject`, a prototype which was added to the object part of COLA (cf. Sec. 2.2). `ProxifiedObject` is implemented in such a way that cloning it results in a pair of two objects: a regular object and a *proxy*, as displayed in Fig. 1. All messages, including the `vtable` message, which returns a dictionary containing all messages understood by an object, are delegated by the proxy to the actual regular object. Hence, understanding, adding and removing messages is always realized using the regular object's virtual method table (VMT), as desired. From now on, `ProxifiedObject` will serve as the prototype for all objects created during the execution of a program written in one of the *j*-based languages from Sec. 3.

Essential operations such as the creation of *class* objects and class *instantiation* can now be implemented as a set of macros, constituting an API which can be regarded as part of the *instruction set* of a virtual machine with dedicated support for composing crosscutting concerns. The abstractions in this kernel API constitute an implementation substrate, effectively providing an execution environment realizing the delegation-based machine model. The provided core instructions are as follows.

`define-class` creates an object representing a class by cloning a *base* class object without copying the latter's methods.

This may be `ProxifiedObject` or a class previously created through `define-class`. The inheritance relationship between the two classes is established once more by means of delegation. The actual class object of the newly created class delegates messages which it can not handle to the proxy

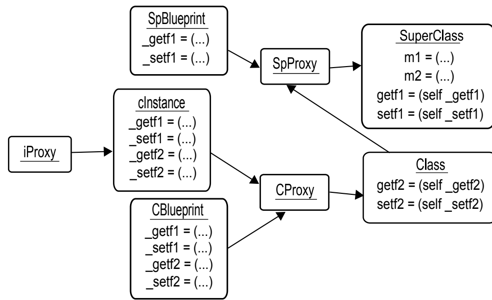

Figure 2: Instance `cInstance` of class `Class`.

of the base class. Hence, all class objects eventually delegate to `ProxifiedObject`, which handles cloning, and ensures classes always appear as a combination of a proxy and the actual class object.

Accessor methods for each field are installed in the class's VMT. Field access should always occur via these class-wide accessors. This is necessary to allow for the possibility of inserting a proxy object in between the class proxy and the actual class object, which may then intercept field access for all instances at once. Next, a *blueprint* object is created, a reference to which is stored in the class object. The blueprint object contains slot-based accessors [14] which must be installed in instance objects. These slot-based accessors essentially manage a memory location for each field and are invoked by the class-level accessors. They are prefixed with `_` to prevent them from hiding the class-level accessors.

`define-send` creates a function which is installed as a method in the VMT of a (class) object. All methods created this way declare a formal *sender* argument, making the caller object of a message available in the corresponding method implementation. The default value for the sender argument is `self` at the caller site.

`create-instance` instantiates a class object by cloning its blueprint object, arranging the delegation chain so that the instance object delegates to the class proxy, as displayed in Fig. 2, and installing the accessors found in the class's blueprint.

`create-proxy` creates a proxy object with the purpose of inserting it in an object's delegation chain, in between that object's proxy and the actual object. A number of methods may be installed in a proxy's VMT. As such, the proxy will effectively intercept those messages before they reach their intended receiver. Additionally, a proxy may install slot-based accessors for fields in its VMT, thus allowing for dynamic field introductions (cf. [6] and Sec. 4.3).

`insert-proxy` inserts a proxy in the delegation chain of a specific object. If no existing proxy is provided, a new one is created as in `create-proxy`.

`remove-proxy` searches an object's delegation chain for a certain proxy and removes it.

`define-proxy-send` creates a function which is installed as a method in a proxy's VMT. The reason a dedicated instruction is necessary besides `define-send` is that a proxy's VMT cannot be obtained by sending the `vtable` message, as a proxy, by definition, does not understand that message.

`send` sends a message to an object. `self` is passed as *sender* argument, unless an alternative is explicitly provided. If the message is not understood, it is delegated to the next object in the delegation chain. This object is found by sending the `_delegate` message to the original receiver.

`resend` may be invoked in a method body, and forces the message to be delegated along the delegation chainwithout rebinding `self`. The sender argument remains unchanged as well, as any receivers further along the delegation chain should be unaware of the fact that the message has been intercepted earlier. Resending is useful, e. g., in case of around advice, where resending a message allows the original method implementation, further along the delegation chain, to be executed.

4.2 The *j* Implementation

Taking another look at Lst. 1, we will now explain how *j*'s language constructs can be mapped onto the kernel API.

For each class, parsed by the `CLS` rule, `define-class` is invoked. As arguments, the class name, the name of its base class (defaulting to `ProxifiedObject`) and a list of field names are passed. For each method, parsed by the `METH` rule, a `define-send` is generated with a formal argument named `x`, installing the method implementation in the class's VMT.

The method body comprises a number of expressions. Constant expressions, as parsed by the `SPECIAL` rule, just map to their equivalent in COLA, while `this` maps to `self` and `x` does not even need translation, as it is known since it was declared a formal argument.

`new` expressions result in a call to `create-instance`. All other variants of the `EXP` rule generate `send`s, be it accessors (in case of field read or write) or method invocations. All instructions generated so far can be regarded as initialization, as they create the necessary objects and install the required functionality. In order to actually execute the program, a final `send` is generated which sends a message called `main` to a command-line specified *entry class*. `main` is always passed an instance of the entry class as an argument, enabling access to its fields via `x`.

Consider the example listed in Lst. 4, which is written in *j* syntax. A first effect of parsing the `Subject` class is the generation of

```
(define-class Subject (attr obs))
```

Next, all three methods are installed in the class's VMT. For `setObserver`, for example, this is done by

```
(define-send setObserver Subject x <impl>)
```

where `<impl>` is the method body, which assigns the value `x` to `self`'s `obs` field by means of

```
(send setobs self x)
```

The `Observer` and `Main` classes are handled similarly. `Main` holds the entry point to the program, being the `main` method. In order to start execution, the `main` message is sent directly to the `Main` class object, while an instance of `Main` is passed as the argument:

```
(send main Subject 0 (create-instance Main))
```

```
1 class Subject {
2   Integer attr
3   Observer obs
4
5   void doSomething(Object x) {
6     this.attr := 25 }
7   void changed (Object x) {
8     this.obs.update(this) }
9   void setObserver (Observer x) {
10    this.obs := x }
11 }
12 class Observer {
13   void update (Subject x) {
14     out.println(x.attr) }
15 }
16 class Main {
17   Subject s
18   Observer o
19
20   void main(Main x) {
21     x.s := new Subject;
22     x.o := new Observer;
23     x.s.setObserver(x.o);
24     x.s.doSomething(null);
25     x.s.changed(null) }
26 }
```

Listing 4: Example *j* code

Note that the extra argument 0 is passed as the sender, as the default value `self` is undefined here. The sender is therefore not available during execution of `main`. The program in Lst. 4 eventually outputs the number 25.

4.3 The *iaj* Implementation

Looking at Lst. 2, *iaj* basically adds aspects, inter-type declarations, advice, a `proceed` expression and a few predefined variables.

For aspects, the `define-aspect` function is implemented on top of the kernel. It is rather similar to `define-class`, but somewhat simpler, since aspect objects are singletons and can not be instantiated. Consequently, they do not need a blueprint object, and slot-based accessors for fields can be installed in the aspect object itself.

A method introduction is handled similarly to a normal method, except that it is installed in the specified class's VMT, and not in the one of the class being defined.

Field introductions are a bit more tricky, as elaborated in [6]. First of all, class-wide accessors for the field are installed in the specified class's VMT. Then, a pair of accessors prefixed with `_`, which would normally be in the blueprint object, is installed class-wide as well, albeit with a specific implementation: instead of containing code for field manipulation, they actually invoke `insert-proxy`, which creates a proxy and inserts it in the delegation chain of the instance which sent the accessor message. The proxy contains the actual slot-based accessors for the field value. Next, the accessor message is sent to `self` once more. This time, however, it will be intercepted by the proxy, and the field value will be read or written, as appropriate. From now on, the class-wide accessors with the same name as the slot-based accessors in the proxy are effectively hidden, ensuring proxy creation for the same field will never occur twice.

Upon parsing non-cflow advice (cf. the `ADVICE` rule), an `insert-proxy` instruction is generated, inserting a proxy in the delegation chain of the class object specified in the pointcut. Next, a method, as specified in the pointcut, is inserted in the proxy's VMT by means of `define-proxy-send`.

In case of cflow advice, the same mechanism should still apply, except that proxy insertion should now occur dynamically upon entering the control flow of a particular method. This is achieved by means of continuous weaving (cw), as explained in [6]. This involves a second proxy (the cw-proxy) which intercepts the cflow method, inserts the required proxy while undeploying itself, then delegates in order to execute the cflow method, and finally removes the proxy and deploys itself again. The cw-proxy needs to dynamically undeploy and redeploy itself in the process, in order to guarantee it will insert only one proxy, even in the advent of recursive calls.[1]

Note that, since field accesses and method invocations are handled by means of message sends, virtually no distinction between `get`, `set` and `call` pointcuts is needed, except in the argument list. Indeed, a getter does not take any arguments, while setters and regular methods take exactly one.

The `proceed` expression, which should only occur in `around` advice, maps nicely to a `resend`, whose result is kept in a local `result` variable. This variable is eventually returned as the return value of the advice. In case of `before` and `after` advice, a `resend` instruction is automatically added, respectively at the end and the beginning of the advice.

Dealing with the additional predefined variables `r`, `s` and `v`, which should only appear inside advice, is straightforward as well. `r` denotes the intended receiver object, which is the instance to which the message was sent. Since the advice proxy was inserted in the delegation chain of this instance's class, and hence also in the instance's delegation chain, this is equivalent to `self`. In normal methods, `this` maps to `self`, but in advice `this` should refer to the aspect singleton, which is easily achieved since aspect names, like class names, are available in the global namespace.

`s` denotes the sender object that sent the message in question. As stated in Sec. 4.1, a formal `sender` argument is always provided by `define-send`, and a value is always passed when using `send`. Therefore `s` can simply be mapped to `sender`. Finally, `v` is only relevant in advice for `set` join points, and denotes the value that was passed to the setter message. As advice are implemented as a method with the same signature as the original method, this value is passed to the advice anyway, and therefore readily available.

Consider the sample *iaj* code in Lst. 5. When the `Observer` aspect is parsed,

```
(define-aspect Observer (changed))
```

is generated, creating a singleton object with slot-based accessors for the `changed` field.

Introduction of the `changed` method into the `Subject` class is handled by generating code in order to insert it in `Subject`'s VMT:

```
(define-send changed Subject x <impl>)
```

Note that this code is indeed exactly the same as in Sec. 4.2, although the method was declared in a different class this time. This means the occurrence of `this` in its body will not refer to the aspect singleton, but to the active instance of `Subject`.

[1] This implementation deviates from the model description [6] which utilizes *delegate functions* whose result depends on the thread. As delegate functions and multi-threading are not yet supported in the AOP kernel implementation, the solution presented here was chosen.

```
1  class Subject {
2    Integer attr
3  }
4  aspect Observer {
5    bool changed
6
7    Subject <-- bool processed
8    Subject <-- void changed(Object x){
9      out.println(this.attr);
10     this.processed := true }
11
12   before set(Integer Subject.attr){
13     this.changed := v.neq(r.attr) }
14   after set(Integer Subject.attr){
15     if (this.changed){ r.changed(null) }
16       else { null }}
17 }
18 aspect Logger {
19   before get(Integer Subject.attr) &&
20          cflow(void Subject.changed(Object)) {
21     out.println("LOG: get Subject.attr") }
22 }
23 class Main {
24   Subject s
25
26   void main (Main x) {
27     x.s := new Subject;
28     x.s.attr := 25 }
29 }
```

Listing 5: Example *iaj* code

The introduction of the `processed` field, which is present in Lst. 5 mainly for illustration purposes, needs a bit more complicated code. First, class-wide accessors for the field are installed in `Subject`'s VMT:

```
(define-send getprocessed Subject
        (send _getprocessed self sender))
(define-send setprocessed Subject value
        (send _setprocessed self sender value))
```

Next, the special accessors are installed as well:

```
(define-send _getprocessed Subject
    (let () (insert-proxy self (processed))
            (send _getprocessed self sender)))
(define-send _setprocessed Subject value
    (let () (insert-proxy self (processed))
            (send _setprocessed self sender value)))
```

Their implementations are very similar. In fact, the first instruction, `insert-proxy`, is exactly the same. It creates a proxy with proper slot-based accessors `_getprocessed` and `_setprocessed`, and inserts it in the delegation chain of `self`, which refers to the (proxy of) the instance object which attempted to access the field. The second part of the method bodies of the special accessors simply sends the relevant accessor message once more to `self`. This time, it will be intercepted by the proxy, and the appropriate action will take place. All this time, the sender argument is simply passed on instead of updated, so that the accessor implementation ultimately handles the original message, is unaware of whatever happened behind the scenes.

The `before` advice on line 12 in Lst. 5, which should apply upon setting the `attr` field of the `Subject` class, triggers creation of a proxy, which is inserted in the delegation chain of the `Subject` class object:

```
(insert-proxy Subject)
```

Next, a method with the same signature as the setter of `attr` is installed in the proxy's VMT:

```
(define-proxy-send setattr <proxy> v <impl>)
```

where `<proxy>` is the name of a local variable to which the result of `insert-proxy` has been assigned, and `<impl>` consists of the instructions in the body. Because the `define-proxy-send` above explicitly declares `v` as a formal parameter, the occurrence of `v` in the body does not even need translation. `r.attr` on the other hand will result in:

```
(send getattr self)
```

Indeed, as the proxy is inserted in the delegation chain of `Subject`, `r` is translated to `self`, which traverses the same delegation chain again, and the appropriate accessor for `attr` will eventually be encountered. The last part of `<impl>` ensures that, once the advice has been executed, the originally intended setter message is sent, by resending the message further along the delegation chain:

```
(resend v)
```

There is a second aspect, called `Logger`, the single advice of which causes a logging statement to be output before getting the value of `Subject.attr`, but only if this occurs within the control flow of `Subject.changed()`. Hence, a continuous weaving proxy is created with around advice for `Subject.changed()`:

```
(insert-proxy Subject)
(define-proxy-send changed <proxy> x
  (let (prx (insert-proxy Subject))
    (define-proxy-send getattr prx <impl>)
    (remove-proxy _self self)
    (resend x)
    (insert-proxy _self self)
    (remove-proxy Subject prx)))
```

The dynamic undeployment and redeployment of the cw-proxy itself is done by means of the two lines surrounding `resend`. Here, `_self` is a pseudovariable available in COLA, which corresponds to the very object in a delegation chain which understands the current message. Since an aspect proxy always implements its own methods, `_self` points exactly to the cw-proxy, and can be used in order to remove it from the delegation chain in which it resides, which is trivially always `self`.

Once more, execution of the code in Lst. 5 will result in the number 25 being printed, preceded by exactly one log statement. The aspect-oriented extensions of the *iaj* language have, however, provided better modularization, as the definition of `Subject` is no longer polluted with implementation details of the Observer pattern. Additionally, the `changed` method no longer needs to be called from the `main` method, but is called automatically by means of the `after` advice. Mapping these aspect-oriented extensions to our kernel API proved to be straightforward and simple, demonstrating the kernel API provides appropriate abstractions.

4.4 The *cj* Implementation

The last of the languages in the *j* family is *cj*. It takes a layer-based approach to achieve modularization of crosscutting concerns. Revisiting the syntax definition in Lst. 3, two major extensions to *j* stand out. On the one hand, the `LAYER` and `CLAYER` rules, which introduce layers, and on the other hand the `withlayer` and `withoutlayer` expressions, which allow for layer activation and deactivation.

Although methods defined in layers are to some extent comparable to around advice, the main difference is that they should not be active by default, but only in the dynamic extent of a `withlayer` block. Therefore, layer methods are

```
1 class Subject {
2   Integer attr
3
4   void setAttr(Integer x) {
5     this.attr := x }
6   void changed(Object x) {
7     out.println(this.attr) }
8 }
9 layer Observer {
10   bool changed
11
12   void Subject.setAttr(Integer x) {
13     thisLayer.changed := x.neq(this.attr);
14     proceed(x);
15     if(thisLayer.changed) { this.changed(null) }
16       else { null } }
17 }
18 class Main {
19   Subject s
20
21   void main (Main x) {
22     x.s := new Subject;
23     withlayer(Observer) {
24       x.s.setAttr(25) } }
25 }
```

Listing 6: Example *cj* code

still translated into proxies, but this time the proxies are not automatically inserted in the delegation chain of the relevant class object. Instead they are stored in *layer objects*, and insertion does not take place until a `withlayer` block is entered. At the end of the block, the inverse takes place and the proxy is removed from the delegation chain again. Conversely, the `withoutlayer` block ensures the specified layer is temporarily deactivated by removing any installed proxies and reinserting them at the end of the block.

In order to support layers, some mechanisms are introduced on top of the kernel API. `define-layer` is similar to `define-aspect`, but adds a field to the resulting layer object, associating proxies generated from methods defined in the layer, with the classes they apply to. Furthermore, layer objects do not initially allocate memory for fields, because layers can be defined in fragments, and hence it is possible additional fields are declared when another fragment of the same layer is parsed. Therefore, layer fields are handled as dynamic field introductions to the singleton layer object, generating code for the creation of a proxy with memory for the field's slot-based accessors, and its insertion in the delegation chain of the layer object.

Handling the `withlayer` and `withoutlayer` is done by the `activate-layer` and `deactivate-layer` functions, which also are implemented on top of the kernel. They take care of respectively insertion and removal of the proxies associated with a certain layer, in the delegation chains of the class objects the proxies are associated with.

The `proceed` instruction is handled in the same way as with `around` advice, resulting in a `resend` instruction. This is not surprising because of the similar treatment of layer methods and `around` advice.

An example of *cj* code can be found in Lst. 6. Parsing the `Observer` layer generates

```
(define-layer Observer)
```

Next, care is taken of the dynamic introduction of the `changed` field into the layer:

```
(insert-proxy Observer (changed))
```

The `setAttr` layer method is translated into `create-proxy`, resulting in a proxy which is not inserted yet, but stored in the layer object and associated with the name `Subject`. The `setAttr` method is then installed in the proxy:

```
(define-proxy-send setAttr Subject x <impl>)
```

The `proceed` instruction in `<impl>`, is transformed to

```
(resend x)
```

while `thisLayer` refers to the `Observer` layer object. Once more, `this` can just be translated into `self` as, upon execution, the proxy will have been inserted in `Subject`'s delegation chain.

Execution of the program in Lst. 6 again results in an output of the number 25. Note how modularization was improved with a layer-based mechanism this time, yet the translation process to our kernel API was as straightforward and simple as with the pointcut and advice flavour.

5. RELATED WORK

Numerous projects have been concerned with the implementation of run-time environment support for MDSOC, and AOP in particular.

The first project to introduce VM-level support for AOP was PROSE [15], which utilized debugger breakpoints as provided by the *Java Virtual Machine Debug Interface* to notify an AOP infrastructure implemented at application level of join point occurrences.

One of the most important driving forces in the development of Steamloom [1, 5] was high performance of woven code. Hence, not all of the join points were instrumented, like in 2nd-generation PROSE [16], nor was a *debugging* infrastructure used for *programming* purposes. Steamloom has followed a just-in-time (JIT) compiler driven approach right from the start. Its fully dynamic weaver, meaning that all weaving steps, including pointcut evaluation, take place at run-time, instruments only those join point shadows known to (maybe) yield join points. Woven code is minimal. Where residual logic [10] is required, it relies on VM-level callbacks for performance. Weaving in Steamloom is achieved through modifying method bytecodes and JIT-recompiling them accordingly.

More recent implementations of the PROSE architecture [11] also feature sophisticated JIT compiler strategies to achieve performance.

The first projects that were described as general-purpose AOP kernels were Reflex [18] and Steamloom [5]. Reflex is based on behavioral reflection and performs load-time preparation of Java classes based on pre-existing specifications; at run-time, prepared join point shadows can be subject to dynamic weaving. Steamloom features an API-based approach to aspect definition that can be targeted by compilers for various languages.

The implementation and language mappings presented in this paper demonstrate how an AOP kernel can be provided at the VM level. Admittedly, Steamloom was the first system to represent both VM support and the AOP kernel idea, but as it ultimately is a Java VM, the applicability of its approach is limited. The same holds for Reflex, which is tied to the JVM platform, and client languages targeting it.

Conversely, the support kernel for implementations of languages supporting the modularization and (possibly dynamic) composition of crosscutting concerns presented in this paper

is not limited with respect to the choice of client language. The underlying execution model's simplicity renders it a viable candidate for wide applicability in the domain of such languages' implementations. Entire VMs can be built on top of it.

The *aspect language implementation architecture* (ALIA) project [2], which is a reference architecture for execution environments for AOP languages, based on a standard execution model, appears to be the only other approach bringing about similar qualities. However, its reliance on a much more sophisticated meta-model may place some limitations on its applicability since a significant modelling effort is required prior to language implementation. Conversely, the AOP kernel presented herein allows for a direct mapping of language mechanisms to lower-level execution mechanisms—more resembling the way a compiler works.

6. SUMMARY AND FUTURE WORK

We have presented an implementation substrate of a machine model, supporting language mechanisms for modularizing crosscutting concerns. We developed a kernel API, provided as an instruction set for a dedicated virtual machine, which served as a compilation target for a number of high-level programming languages which include mechanisms addressing crosscutting concerns. We showed that a large subset of relevant language constructs can be mapped onto this instruction set in a simple and straightforward manner. As a broad variety of constructs, ranging from aspects, advice and pointcuts to dynamically scoped and activated layers were included, we conclude that the machine model indeed provides a compelling translation target for languages supporting different programming styles and modularization features for crosscutting concerns.

Future work will put a strong focus on performance, the goal being to demonstrate that execution of our instruction set can be done in an efficient manner. In fact, the nature of the machine model gives multiple opportunities to develop *dedicated* optimization strategies for MDSOC language implementations—especially regarding object communication through messages, and memory management. For example, dedicated caching mechanisms may optimize message sending and delegation chain modifications. Dedicated garbage collection schemes, on the other hand, might take advantage of the large number of small objects employed in the model.

7. ACKNOWLEDGEMENTS

The authors are grateful to Ian Piumarta, for valuable feedback on the COLA framework.

8. ADDITIONAL AUTHORS

Dirk Janssens (University of Antwerp, Belgium,
email: dirk.janssens@ua.ac.be)

9. REFERENCES

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

[2] C. Bockisch, M. Mezini, W. Havinga, L. Bergmans, and K. Gybels. Reference Model Implementation. Tech. Report AOSD-Europe deliv. D96, AOSD-Europe-TUD-8, TU Darmstadt, August 2007.

[3] B. Ford. Parsing Expression Grammars: a Recognition-based Syntactic Foundation. *SIGPLAN Not.*, 39(1):111–122, 2004.

[4] H. Schippers, D. Janssens, M. Haupt, and R. Hirschfeld. Delegation-Based Semantics for Modularizing Crosscutting Concerns. In *OOPSLA 2008, Nashville, TN, USA, oct 19 - oct 23*, 2008.

[5] M. Haupt. *Virtual Machine Support for Aspect-Oriented Programming Languages*. PhD thesis, Darmstadt University of Technology, 2006.

[6] M. Haupt and H. Schippers. A machine model for aspect-oriented programming. In *ECOOP 2007, Berlin, Germany*, volume 4609 of *LNCS*, pages 501–524. Springer, 2007.

[7] R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-oriented Programming. *Journal of Object Technology (JOT)*, 7(3):125–151, March-April 2008.

[8] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In M. Aksit and S. Matsuoka, editors, *ECOOP '97*, volume 1241 of *LNCS*, pages 220–242. Springer, 1997.

[9] H. Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In *OOPSLA 1986*, pages 214–223. ACM Press, 1986.

[10] H. Masuhara, G. Kiczales, and C. Dutchyn. A Compilation and Optimization Model for Aspect-Oriented Programs. In G. Hedin, editor, *CC 2003*, volume 2622 of *LNCS*, pages 46–60. Springer, 2003.

[11] A. Nicoara, G. Alonso, and T. Roscoe. Controlled, Systematic, and Efficient Code Replacement for Running Java Programs. In *Proc. EuroSys 2008, Glasgow, UK.* ACM 2008.

[12] H. Ossher. A direction for research on virtual machine support for concern composition. In *Proc. Workshop VMIL '07*. ACM Press, 2007.

[13] I. Piumarta. Accessible Language-Based Environments of Recursive Theories. Technical Report VPRI Research Note RN 2006-001-a, Viewpoints Research Institute, 2006.

[14] I. Piumarta and A. Warth. Open, Reusable Object Models. Technical Report VPRI Research Note RN 2006-003-a, Viewpoints Research Institute, 2006.

[15] A. Popovici, T. Gross, and G. Alonso. Dynamic Weaving for Aspect-Oriented Programming. In G. Kiczales, editor, *AOSD 2002*. ACM Press, 2002.

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

[17] M. C. Skipper. *Formal Models for Aspect-Oriented Software Development*. PhD thesis, Imperial College, London, 2004.

[18] E. Tanter, J. Noyé, D. Caromel, and P. Cointe. Partial Behavioral Reflection: Spatial and Temporal Selection of Reification. In *OOPSLA 2003*. ACM Press, 2003.

[19] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *ICSE 1999*, pages 107–119. ACM Press, 1999.