3-31-2010

Towards Efficient Java Virtual Machine Support for Dynamic Deployment of Inter-type Declarations

Bashar Gharaibeh
Iowa State University

Hridesh Rajan
Iowa State University

J. Morris Chang
Iowa State University

Towards Efficient Java Virtual Machine Support for Dynamic Deployment of Inter-type Declarations

Bashar Gharaibeh, Hridesh Rajan, and J. Morris Chang

TR #09-23b
Initial Submission: October 9, 2009.
Revised: March 31, 2010.

Keywords: Inter-type declarations, invocation, weaving, aspect-oriented intermediate-languages, Nu, aspect-oriented virtual machines
CR Categories: D.1.5 [*Programming Techniques*] Object-oriented Programming D.3.3 [*Programming Languages*] Language Constructs and Features D.3.4 [*Programming Languages*] Processors

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1041, USA

Towards Efficient Java Virtual Machine Support for Dynamic Deployment of Inter-type Declarations

Bashar Gharaibeh Hridesh Rajan J. Morris Chang
Iowa State University
{bashar, hidesh, morris}@iastate.edu

Abstract

Dynamic deployment is an important feature of an aspect-oriented language design that has many applications, e.g. in runtime monitoring, runtime adaptation to fix bugs or add features to long running applications, runtime update of dynamic policy changes, etc. Many recently proposed language designs support these use cases. In previous work, researchers have demonstrated that the ability to support unanticipated deployment enables simpler and often more efficient implementations. These works have addressed an important subset of aspect-oriented features namely those that can be represented as the pointcut-advice model. In this work, we describe the design, formal semantics, and implementation of our strategy for efficiently supporting dynamic deployment of inter-type declarations, which is another important aspect-oriented feature. Additional contributions of this work are: a detailed real world case study that demonstrates the need for supporting dynamic deployment of inter-type declaration for online update of long running applications, and a prototype based on the Jikes Research Virtual Machine that supports these features. We evaluate our prototype via a rigorous performance analysis, which shows that flexible, dynamic deployment of inter-type declarations can be efficiently supported in a Java virtual machine.

1. Introduction

Dynamic deployment of aspect-oriented features has many applications, e.g. in runtime monitoring [7], runtime adaptation to fix bugs or add features to long running applications [48], runtime update of dynamic policy changes [38, 39], etc. Driven by these use cases, dynamic aspect-oriented constructs [3, 48] have received significant attention in recent aspect-oriented literature [2, 5, 8, 12, 17, 20, 38, 45].

Driven by the popularity of dynamic aspect-oriented features, we set out to study the effectiveness of aspect-orientation in enabling dynamic deployment of features in long-running applications. For such applications, disruption can be a cause of major customer annoyance and/or large business loss [30, 36, 37]. Thus, we believe that such a use case could be a killer application of aspect-oriented software development (AOSD). As we discuss in Section 4, our belief turned out to be quite true. Our challenge was to update the Apache's Xerces XML parser [50] library version from 1.2.3 to 1.3.1. Being able to update without taking down the web server appeared to be a lucrative proposition for this long running application. We wanted to express the changes between versions as aspects and apply them dynamically as *modular units of software update*.

During the course of our study, however, we met with a roadblock. Much of the existing work on dynamic deployment so far (including our own [12]) has focused only on the pointcut-advice (PCA) model [32]. For our use case, more often than not we found ourselves expressing changes as inter-type declarations (ITDs). Emulating inter-type declarations using pointcut advice model is feasible, but such emulation adds unnecessary design and runtime complexity. Haupt and Schippers's delegation-based model [17, 45] and Kuhn and Nierstrasz's object-fragment model [27] seemed capable of expressing these features, however, efficient implementation of neither approaches has been demonstrated as of this writing. Efficiency was of paramount importance for our use case. A web-server capable of dynamically updating itself, but running an order of magnitude slower would not be a pragmatic proposition.

Driven by this important use case for AOSD, we set out to develop efficient support for flexible, dynamic deployment of ITDs. For dynamic updating, it is equally important to preserve the correctness of the long-running application. To that end, we developed a formal foundation for our approach of supporting dynamic deployment. Two benefits accrued from that. First, formal modeling allowed us to systematically determine the changes that would be needed in the runtime representation of the application in a virtual machine, and second, it allowed us to identify the precise condi-

tions on safe updating of runtime applications using dynamically deployed inter-type declarations that are independent of a specific virtual machine design. This formalization presented in Section 2 also helps us illustrate the key ideas and the intended basic semantics.

We then developed an efficient implementation for dynamic deployment of ITDs in the JikesRVM [22], which we describe in Section 3. The prototype enabled us to express the desired update of the Xerces XML parser as a total of 46 features, where only five features can not be modeled as inter-type declaration or pointcut advice aspects (Section 4). We also measured the performance of our prototype using DaCapo and SPECjvm98 benchmarks, which shows negligible steady-state performance overhead and acceptable overheads during deployment of aspect-oriented features for the flexibility that it provides (Section 5). In summary, we contribute:

- A formal description of the deployment process of an inter-type declaration;
- A prototype of our approach in JikesRVM;
- A relatively large-scale case study that shows its utility towards dynamic updating; and
- A rigorous performance evaluation of our prototype.

The rest of this paper describes our approach for dynamic deployment of ITDs.

2. On Dynamic Introductions

We first present our main ideas for supporting dynamic inter-type declarations via a core object-oriented calculus, which helps elucidate the major challenges in the design of its semantics. Readers not interested in formal details and soundness proof may skip Sections 2.3-2.5.

Our formalization of dynamically deployed inter-type declarations helps us distinguish between safe and unsafe ordering of dynamic deployment of inter-type declarations. Our soundness result in that regard is that for every *statically well-typed* inter-type declaration, there exists a safe ordering for its dynamic deployment.

2.1 Basics: Abstract Syntax

To illustrate key ideas in the semantics of our strategy for supporting dynamic inter-type declarations, we abstract away from the details of a typically very complex VM structure and full-fledged Java bytecode language [16]. Instead, we choose to describe our ideas using an OO calculus from Clifton's work[10] that is similar to Classic Java [14] and Featherweight Java [21]. Unlike Haupt and Schippers's model [17, 45], which needs support for delegation in the program configuration and thus uses the δ calculus [1], an OO calculus suffices for our runtime model. Our calculus has classes, objects, inheritance, and subtyping, but it does not have **`super`**, interfaces, exception handling, built-in value types, privacy modifiers, or abstract methods. As we show in Section 3, omission of these features does not affect our discussion primarily because the challenges in supporting dynamic inter-type declarations are in updating classes, their fields and methods. These core features are present in our calculus. In addition, we explicitly do not model aspect-oriented constructs such as advice and pointcuts primarily because dynamic deployment of these features is already discussed elsewhere, e.g. [5, 12]. The abstract syntax of the calculus is shown in Figure 1.

Abstract Syntax:

$$
\begin{array}{l}
\mathit{prog} ::= \overline{\mathit{decl}}\ e \\
\mathit{decl} ::= \texttt{class}\ c\ \texttt{extends}\ d\ \{\ \overline{\mathit{field}}\ \overline{\mathit{meth}}\ \} \\
\mathit{field} ::= t\ f\texttt{;} \\
\mathit{meth} ::= t\ m\ (\ \overline{\mathit{form}}\ )\ \{\ e\ \} \\
t ::= c \\
\mathit{form} ::= t\ \mathit{var},\ \text{where } \mathit{var} \neq \texttt{this} \\
e ::= \texttt{new}\ c\,() \mid \mathit{var} \mid \texttt{null} \mid e.m\,(\overline{e}) \\
\quad \mid e.f \mid e.f = e \mid \texttt{cast}\ c\ e \mid e\ \texttt{;}\ e
\end{array}
$$

where

c, d	$\in$	$\mathcal{C}$, the set of class names
f	$\in$	$\mathcal{F}$, the set of field names
m	$\in$	$\mathcal{M}$, the set of method names
var	$\in$	$\{\texttt{this}\} \cup \mathcal{V}$, $\mathcal{V}$ is the set of variable names

Figure 1. Abstract Syntax.

The technical description of this calculus builds on our previous work [42] and other work on OO calculus [10, 14, 21]. It contains a single top-level form for classes and common expressions for the construction of an object (**`new`** $c\,()$), variable dereference (*var*, including **`this`**), field dereference ($e.f$), **`null`**, cast (**`cast`** $t\ e$), assignment to a field ($e_1.f = e_2$), and sequencing (e_1**`;`** e_2). Their semantics and typing is fairly standard [10, 14, 21]. A program consists of a sequence of declarations ($\overline{decl}$) followed by an expression. This expression is like the "main" method. The overline notation is used throughout this paper to represent sequences. All declarations in a program are formed into a fixed list CT (i.e. Class Table), which is then used in giving semantics of expressions [14].

2.2 Basics: Inter-type Declarations

With the basic ideas for a core object-oriented language in place, we now turn to a small extension necessary to syntactically define inter-type declarations. As mentioned previously, we omit the pointcut-advice model in this work as they have been extensively covered elsewhere [5, 6, 11]. There is no novelty in the syntax, rather it is a straightforward adaptation from other AO languages [23, 44]. These extensions are presented in Figure 2. Inter-type declarations can be used to make many type of changes to the structure of a program. Among these most common changes are adding a field to an existing class, adding a method to an existing class and adding a parent to an existing class to add an additional interface. There are other features that languages like AspectJ [23] support, which allow developers to encode statically-checked constraints into their program,

e.g. the `declare error` construct. However, such features are useful only during compilation, thus, there are no reasons to support them in a dynamic deployment system for inter-type declarations.

decl ::= ... | **itd** *i* { $\overline{\textit{asuper}}$ $\overline{\textit{afield}}$ $\overline{\textit{ameth}}$ }
asuper ::= *c* **extends** *d*;
afield ::= *t c* . *f*;
ameth ::= *t c* . *m* ($\overline{\textit{form}}$) { *e* }

where
i ∈ $\mathcal{I}$, the set of ITD names

Figure 2. Abstract Syntax of Inter-type Declarations (`itd`).

The syntax is self-explanatory and provides support for extending the class hierarchy by adding a superclass, adding a field to a class and adding a method to a class. The non-terminals not defined in this figure are the same as in Figure 1. We require that fully-qualified names of added fields are distinct and that *asuper*, *afield*, and *ameth* declarations appear in this order inside inter-type declarations. An example appears below that adds two fields and two methods to the `Node` class. These help keep track of the number of sent and received packets.

```
class Node extends Object {
  Integer address;
  Integer receive(Packet p){...}
  Integer Packet send(){...}
}

itd Statistics{
  Integer Node.numSent; Integer Node.numRcvd;
  Integer Node.incSent(){numSent.incr()}
  Integer Node.incRcvd(){numRcvd.incr()}
}
```

A static deployment model for inter-type declarations would merge these declarations into object-oriented declarations to produce a pure object-oriented program expressed in terms of the abstract syntax shown in Figure 1. So the fields and methods declared in the `Statistics` inter-type declaration will be directly inserted in the class `Node`. In a dynamic deployment model, however, we would like to keep these two representations separate. To that end, we can produce an operational representation from the declarative representation expressed using the rules in Figure 2. This could be thought of as the intermediate representation of the inter-type declarations, which would be run by the dynamic deployment infrastructure to modify the program's structure at runtime.

te ::= **addf** (c, t, f)
| **addm** $(c, t, m, \overline{\textit{form}}, e)$
| **addp** (c, d)
| **replm** $(c, t, m, \overline{\textit{form}}, e)$

Figure 3. Dynamic Deployment Expressions.

We have identified that most inter-type declarations could be expressed using four expressions shown in Figure 3. The non-terminals not defined in this figure are the same as in Figure 1. The **addf** $(c,\ t,\ f)$ expression adds a field f of type t to the class named c. The **addm** $(c,\ t,\ m,\ t_1\ var_1, \ldots, t_n\ var_n,\ e)$ expression adds a method with return type t, body e and formal parameters $t_1\ var_1, \ldots, t_n\ var_n$ to the class named c. The **addp** $(c,\ d)$ expression adds an existing class d as the super class of the class named c. The **replm** $(c,\ t,\ m,\ t_1\ var_1, \ldots, t_n\ var_n,\ e)$ expression replaces the body of the method with return type t and arguments $t_1\ var_1, \ldots, t_n\ var_n$ in the class named c with e. This distinction between adding and replacing a method is to simplify treatment of the *ameth* declarations in the semantics.

2.3 Type Checking of Inter-type Declarations

We state the type checking rules using a fixed class table (list of declarations CT) [10, 42]. The class table can be thought of as an implicit inherited attribute used by the rules and auxiliary functions. We require that top-level names in the program are distinct and that the inheritance relation on classes is acyclic. The typing rules for expressions use a simple type environment, Π, which is a finite partial mapping from locations loc or variable names var to a type. The notation $\nu' <: \nu$ means ν' is a subtype of ν. It is the reflexive-transitive closure of the declared subclass relationships.

We then define the rules for transformation from the declarative representation to an operational representation as type checking rules (Figure 4). These rules check the correctness of the inter-type declaration with respect to the program's current list of declarations CT and produce a sequence of deployment expressions represented as $\overline{te}$. The judgment takes the form of $CT \vdash l$ **:** $te; \overline{te}$ where l is the declarative ITD list presenting parent, field and method additions declared inside the **itd**. The term $te; \overline{te}$ represents the list of deployment expression where te is the expression corresponding to the declaration expression being checked. We use the $x :: y$ notation to describe concatenation of elements inside l. In these rules, $\overline{\textit{asuper}}$ denotes a sequence of *asuper* declarations and $\overline{te}$ a sequence of dynamic deployment expressions defined in Figure 3.

The rules for adding parent, field, and method check the standard correctness conditions for addition of the inter-type declaration to the existing program. These rules recursively compute deployment expression, where CT_2 is used as the context for checking remaining declaration expressions. For example, the rule (T-ASUPER) used to generate the deployment expression **addp** (c, d) checks whether c and d are existing classes in the program, and that adding d as the superclass of c will not create an inheritance cycle by stating that d should not be a subtype of c. The next line in rule (T-ASUPER) checks whether the fields and methods of d do not conflict with existing fields and methods of c and its subclasses. If these conditions are satisfied, then we construct a new class table (CT_2) by changing the mapping between class c and its definition in the original class table (CT_1) to a definition where class d is the parent of c. The new class

(T-ITD)

$$\frac{CT_1 \vdash \mathit{asuper}_1 :: \ldots :: \mathit{asuper}_p :: \mathit{afield}_1 :: \ldots :: \mathit{afield}_q :: \mathit{ameth}_1 :: \ldots :: \mathit{ameth}_r \ \texttt{:}\ \overline{te}}{CT_1 \vdash \texttt{itd}\{\mathit{asuper}_1, \ldots,\ \mathit{asuper}_p, \mathit{afield}_1, \ldots,\ \mathit{afield}_q, \mathit{ameth}_1, \ldots,\ \mathit{ameth}_r\}\ \texttt{:}\ \overline{te}}$$

(T-ASUPER)

$$\frac{\begin{array}{c} CT_1(c) = \texttt{class}\ c\ \texttt{extends}\ \texttt{Object}\{\overline{field}, \overline{meth}\} \qquad d \in dom(CT_1) \qquad d \not<: c \\ \forall c' \in \{c' \mid c' \in dom(CT_1) \wedge c' <: c\} \cdot (dom(fieldsOf(c')) \cap dom(fieldsOf(d)) = \emptyset \wedge \\ \forall meth \in methodsOf(c') \cdot override(meth, d)) \qquad CT_2 = CT_1 \oplus \{c \mapsto \texttt{class}\ c\ \texttt{extends}\ d\{\ldots\}\} \qquad CT_2 \vdash \overline{\mathit{asuper}} :: \overline{\mathit{afield}} :: \overline{\mathit{ameth}}\ \texttt{:}\ \overline{te} \end{array}}{CT_1 \vdash\ c\ \texttt{extends}\ d :: \overline{\mathit{asuper}} :: \overline{\mathit{afield}} :: \overline{\mathit{ameth}}\ \texttt{:}\ \texttt{addp}(c, d)\,;\overline{te}}$$

(T-AFIELD)

$$\frac{\begin{array}{c} CT_1(c) = \texttt{class}\ c\ \texttt{extends}\ d\{t_1\ f_1, \ldots, t_n\ f_n, \overline{meth}\} \qquad \forall c' \in \{c' \mid c' \in dom(CT_1) \wedge c' <: c\} \cdot f \notin dom(fieldsOf(c')) \\ t \in dom(CT_1) \qquad CT_2 = CT_1 \oplus \{c \mapsto \texttt{class}\ c\ \texttt{extends}\ d\{t\ f, t_1\ f_1, \ldots, t_n\ f_n, \overline{meth}\}\} \qquad CT_2 \vdash \overline{\mathit{afield}} :: \overline{\mathit{ameth}}\ \texttt{:}\ \overline{te} \end{array}}{CT_1 \vdash\ t\ c\ .\ f :: \overline{\mathit{afield}} :: \overline{\mathit{ameth}}\ \texttt{:}\ \texttt{addf}(c, t, f)\,;\overline{te}}$$

(T-AMETHOD-NEW)

$$\frac{\begin{array}{c} CT_1(c) = \texttt{class}\ c\ \texttt{extends}\ d\{\overline{field}, meth_1, \ldots, meth_n\} \qquad \bullet = lookup(c, m) \qquad \mathit{meth} = t\ m\ (t_1\ var_1, \ldots, t_n\ var_n)\{e\} \\ CT_2 = CT_1 \oplus \{c \mapsto \texttt{class}\ c\ \texttt{extends}\ d\{\overline{field}, meth_1, \ldots, meth_n, meth\}\} \qquad CT_2 \vdash \mathit{meth} : \text{OK in } c \qquad CT_2 \vdash \overline{\mathit{ameth}}\ \texttt{:}\ \overline{te} \end{array}}{CT_1 \vdash t\,c.m(t_1\ var_1, \ldots, t_n\ var_n)\{e\} :: \overline{\mathit{ameth}}\ \texttt{:}\ \texttt{addm}(c, t, m, t_1\ var_1 \ldots t_n\ var_n, e)\,;\overline{te}}$$

(T-AMETHOD-OVERRIDE)

$$\frac{\begin{array}{c} CT_1(c) = \texttt{class}\ c\ \texttt{extends}\ d\{\overline{field}, meth_1, \ldots, meth_n\} \\ (c', t\ m(t_1\ var_1, \ldots, t_n\ var_n)\{e'\}) = lookup(c, m) \qquad c' \neq c \qquad \mathit{meth} = t\ m\ (t_1\ var_1, \ldots, t_n\ var_n)\{e\} \\ CT_2 = CT_1 \oplus \{c \mapsto \texttt{class}\ c\ \texttt{extends}\ d\{\overline{field}, meth_1, \ldots, meth_n, meth\}\} \qquad CT_2 \vdash \mathit{meth} : \text{OK in } c \qquad CT_2 \vdash \overline{\mathit{ameth}}\ \texttt{:}\ \overline{te} \end{array}}{CT_1 \vdash t\,c.m(t_1\ var_1, \ldots, t_n\ var_n)\{e\} :: \overline{\mathit{ameth}}\ \texttt{:}\ \texttt{addm}(c, t, m, t_1\ var_1 \ldots t_n\ var_n, e)\,;\overline{te}}$$

(T-AMETHOD-REPLACE)

$$\frac{\begin{array}{c} CT_1(c) = \texttt{class}\ c\ \texttt{extends}\ d\{\overline{field}, meth_1, \ldots, t\ m\ (t\ m(t_1\ var_1, \ldots, t_n\ var_n)\{e'\}, \ldots, meth_n\} \qquad \mathit{meth} = t\ m\ (t_1\ var_1, \ldots, t_n\ var_n)\{e\} \\ CT_2 = CT_1 \oplus \{c \mapsto \texttt{class}\ c\ \texttt{extends}\ d\{\overline{field}, meth_1, \ldots, meth, \ldots, meth_n\}\} \qquad CT_2 \vdash meth : \text{OK in } c \qquad CT_2 \vdash \overline{\mathit{ameth}}\ \texttt{:}\ \overline{te} \end{array}}{CT_1 \vdash t\,c.m(t_1\ var_1, \ldots, t_n\ var_n)\{e\} :: \overline{\mathit{ameth}}\ \texttt{:}\ \texttt{replm}(c, t, m, t_1\ var_1 \ldots t_n\ var_n, e)\,;\overline{te}}$$

Figure 4. Type-checking rules for inter-type declarations.

table CT_2 is used to check the reminder of the inter-type declaration. Note that this rule assumes the class c did not have a parent except the top type `Object`. A refined version that allows strengthening of parent is also possible with few modifications to this rule. In cases where class c has a parent d, a deployment expression of the form **addp** (c, d') can type-check by adding the condition that $d' <: d$.

There are three rules corresponding to *ameth* that apply depending on whether the method being added does not exist in the class or its superclass (T-AMETHOD-NEW), it exists in the superclass (T-AMETHOD-OVERRIDE), or whether it exists in the current class (T-AMETHOD-REPLACE). Finally, the rule (T-ITD) establishes an ordering on the output deployment expressions and constraints on the type environments used for type-checking various parts of the inter-type declarations expressed in the rule's hypothesis.

2.4 Dynamic Deployment as Configuration and Class Table Transformation

Dynamic deployment of inter-type declarations transform the runtime state of an object-oriented program. We model this state as configuration in an OO small step operational semantics. Programs semantics is provided by giving a semantics of expressions. In the expression semantics we also rely on an expression: *loc*, which represents locations in the store.

The small steps taken in the semantics are defined as transitions from one configuration to another. These configurations are shown in Figure 5. A configuration consists of the current expression e and the store μ. The store is a mapping from locations to object records. An object record (o) is the representation of an object in the store. First, the class name (c) is kept around, so that we may be able to look up the class representation from the class table (CT). Second, the value of each field is kept as map (F) from the field name to its value. The object record allows us to call methods with that object as a receiver and to access the fields of the object and modify their values. These steps are not crucial to the following discussion, but presented in Section A for completeness along with the type checking rules for OO expressions.

Domains:

$\Gamma ::= \langle e, \mu \rangle$	"Configurations"
$\mu ::= \{loc_k \mapsto o_k\}_{k \in K}$, **where** K is finite	"Stores"
$v ::= loc \mid \texttt{null}$	"Values"
$o ::= [c\,.\,F]$	"Object Records"
$F ::= \{f_k \mapsto v_k\}_{k \in K}$, **where** K is finite	"Field Maps"

Figure 5. Domains [10, 14, 21, 42].

Given a statically valid inter-type declaration and the current class table CT_1, the type-checking rules (that can also be thought of as compilation rules) in Figure 4 produce a safe sequence of dynamic deployment expressions ($\overline{te}$). Note

Transformation relation: $\langle CT_1, te :: \overline{te}, \gamma\rangle \rightsquigarrow \langle CT_2, \overline{te}, \gamma'\rangle$

(ADD PARENT)

$$\frac{CT_1(c) = \texttt{class}\ c\ \texttt{extends}\ \texttt{Object}\{\overline{field}, \overline{meth}\} \quad CT_2 = CT_1 \oplus \{c \mapsto \texttt{class}\ c\ \texttt{extends}\ d\{\ldots\}\} \quad \gamma = \langle e, \mu\rangle \quad F' = \{f \mapsto \texttt{null} \mid f \in \mathit{fieldsOf}(d)\} \quad \mu' = \mu \oplus \{\mathit{loc} \mapsto [c'.F \oplus F'] \mid \mathit{loc} \in \mathit{dom}(\mu) \wedge \mu(\mathit{loc}) = [c'.F] \wedge c' <: c\} \quad \gamma' = \langle e, \mu'\rangle}{\langle CT_1, \texttt{addp}\,(c, d)\,;\overline{te}, \gamma\rangle \rightsquigarrow \langle CT_2, \overline{te}, \gamma'\rangle}$$

(ADD FIELD)

$$\frac{CT_1(c) = \texttt{class}\ c\ \texttt{extends}\ d\{t_1\ f_1, \ldots, t_n\ f_n, \overline{meth}\} \quad CT_2 = CT_1 \oplus \{c \mapsto \texttt{class}\ c\ \texttt{extends}\ d\{t\ f, t_1\ f_1, \ldots, t_n\ f_n, \overline{meth}\}\} \quad \gamma = \langle e, \mu\rangle \quad \mu'' = \{\mathit{loc} \mapsto [c'.F \oplus \{f \mapsto \texttt{null}\}] \mid [c'.F] = \mu(\mathit{loc}) \wedge c' <: c\} \quad \mu' = \mu \oplus \mu'' \quad \gamma' = \langle e, \mu'\rangle}{\langle CT_1, \texttt{addf}\,(c, t, f)\,;\overline{te}, \gamma\rangle \rightsquigarrow \langle CT_2, \overline{te}, \gamma'\rangle}$$

(ADD METHOD)

$$\frac{CT_1(c) = \texttt{class}\ c\ \texttt{extends}\ d\{\overline{field}, meth_1, \ldots, meth_n\} \quad meth = t\ m\ (t_1\ var_1, \ldots, t_n\ var_n)\{e\} \quad CT_2 = CT_1 \oplus \{c \mapsto \texttt{class}\ c\ \texttt{extends}\ d\{\overline{field}, meth_1, \ldots, meth_n, meth\}\}}{\langle CT_1, \texttt{addm}\,(c, t, m, t_1\ var_1, \ldots, t_n\ var_n, e)\,;\overline{te}, \gamma\rangle \rightsquigarrow \langle CT_2, \overline{te}, \gamma\rangle}$$

(REPLACE METHOD BODY)

$$\frac{CT_1(c) = \texttt{class}\ c\ \texttt{extends}\ d\{\overline{field}, meth_1, \ldots, t\ m\ (\ \overline{form}\ )\{e'\}, \ldots, meth_n\} \quad meth = t\ m\ (t_1\ var_1, \ldots, t_n\ var_n)\{e\} \quad CT_2 = CT_1 \oplus \{c \mapsto \texttt{class}\ c\ \texttt{extends}\ d\{\overline{field}, meth_1, \ldots, meth, \ldots, meth_n\}\}}{\langle CT_1, \texttt{replm}\,(c, t, m, t_1\ var_1, \ldots, t_n\ var_n, e)\,;\overline{te}, \gamma\rangle \rightsquigarrow \langle CT_2, \overline{te}, \gamma\rangle}$$

Figure 6. Rules for Transforming the class table CT and configuration γ for each deployment expression in Figure 3.

that the deployment of this sequence is valid if and only if the current class table of the running program is CT_1 and the expressions in the deployment sequence are applied in exactly that order. The deployment is then defined as the transformation relation $\rightsquigarrow$ as shown in Figure 6 that given a transformation configuration $\langle CT_1, te\,;\overline{te}, \gamma\rangle$ produces a new transformation configuration $\langle CT_2, \overline{te}, \gamma'\rangle$. Here, CT_1 is the program's current list of declarations, γ its current runtime configuration, CT_2 the new list of declarations, and γ' the new configuration. The final deployment configuration is $\langle CT_1, \bullet, \gamma\rangle$. These rules depend on the static type rules stated previously to check correctness conditions.

The rule (ADD FIELD) says to first find the class c in the current class table and construct a new class table, where the class definition is extended with the new field f. Furthermore, a new store μ' is constructed in which the object record for each object of type c or its subtypes is extended to include a field f, which is initially assigned value **`null`**. The rule (ADD METHOD) simply looks up the class in the current class table and produces a new class table, where the class definition is extended with the new method.

The rule (ADD PARENT) creates a new class table which contains the modified class definition. A new store μ' is also constructed, where the object records for each object of type c or c', where c' is subtype of c is extended to add the new fields from d. These new fields are assigned value **`null`**. Note that due to the requirement of having a statically valid inter-type declaration, we can avoid having multiple parents to the same class. Finally, the rule (REPLACE METHOD BODY) constructs a new class table, where the original method body is replaced by the new body. This is needed to represent ITDs that add methods already existing in the target class.

2.5 Soundness of Dynamic Deployment

The proof of soundness of object-oriented type-system and semantics uses a standard preservation and progress argument [21, 49]. The details are adapted from [10, 42]. The key definition of environment-store consistency follows:

DEFINITION 2.1. *[Environment-Store Consistency.] Let* Π *be a type environment and* μ *a store. Then* μ is consistent with Π*, written* $\mu \approx \Pi$*, if and only if all the following hold:*

1. *For all loc such that* $\mu(loc) = [t.F]$*, loc has type* t *in* Π *and* $dom(F) = dom(\mathit{fieldsOf}(t))$*, and for all loc in* $rng(F)$ *either loc is in store* μ *or loc is null, and for all* $f \in dom(F)$ *such that* $F(f) = loc'$ *and* $\mathit{fieldsOf}(t)(f) = u$ *and* $\mu(loc') = [t'.F']$ *implies* $t' <: u$*.*
2. *Every loc that is in* $dom(\Pi)$ *is also in* $dom(\mu)$ *and* $dom(\mu) \subseteq dom(\Pi)$*.*

With the key defenition of consistency in place, Theorem 2.2 states our main soundness result. It says that given an inter-type declaration (**`itd`** $i\ \{\ldots\}$) and the current class table (CT_1), if i is well-typed then that inter-type declaration type checks to a safe sequence of deployment expressions ($\overline{te}$). We show that for each deployment expression, the resulting class table and configuration are consistent. The soundness of the inter-type declaration follows from the consistency of intermediate configurations.

THEOREM 2.2. *[Safe Deployment.] Let* $\langle e, \mu\rangle$ *be a program configuration where* CT_1 *is the class table for that program,* e *is a well-typed expression and* μ *is a store. Let* Π *be a type environment where each declaration in* CT_1 *type checks and that* $\mu \approx \Pi$ *according to the Definition 2.1. Let* **`itd`** $i\ \{\ldots\}$ *be an inter-type declaration such that* $CT_1 \vdash$ **`itd`** $i\ \{\ldots\}$ **`:`** $te :: \overline{te}$ *and* $\langle CT_1, te :: \overline{te}, \gamma\rangle \rightsquigarrow \langle CT_2, \overline{te}, \gamma'\rangle$*, where* γ *is* $\langle e, \mu\rangle$*,* γ' *is* $\langle e, \mu'\rangle$*,* CT_2 *is* $CT_1 \oplus CT'$*, and* CT' *is the set of modified classes by*

te, and let Π' *be a type environment where each declaration in* CT_2 *type checks and that for all* $loc \in dom(\Pi)$, $\Pi'(loc) = \Pi(loc)$. *Then* $\mu' \approx \Pi'$.

Proof Sketch: The theorem trivially holds for an empty inter-type declaration. For non-empty i notice that an inter-type declaration can have finite number of *asuper*, *afield*, and *ameth* declarations and that type-checking rules for inter-type declaration produce one deployment expression for each *asuper*, *afield*, and *ameth* declaration according to rules in Figure 4. Thus, $\overline{te}$ is finite. Let $\overline{te}$ be $te_0; \ldots; te_{k-1}; te_k; \ldots; te_n$. The rest of the proof is by induction on the cases of te_k.

Case **addf**: If te_k is $\texttt{addf}\,(c,t,f)$, then by the hypothesis of the rule (ADD FIELD) in Figure 6, we have that $CT_2 = CT_1 \oplus \{c \mapsto \texttt{class}\, c\ \texttt{extends}\ d\ \{t\ f, t_1\ f_1, \ldots, t_n\ f_n, \overline{meth}\}\}$ and for every $c' <: c$ and for each $loc \in dom(\mu)$ such that $\mu(loc) = [c'.F]$, $\mu'(loc) = [c'.F \oplus \{f \mapsto \texttt{null}\}]$. Since each declaration in CT_2 type checks in Π', $CT_2(c)$ type checks correctly. Since type checking of t is part of c's type checking, we know that t must be a valid type. Also, for all such object records pointed to by loc, $f \mapsto \texttt{null}$. By rule (T-NULL) for type-checking OO expressions we can take **null** to be of type t, all consistency conditions for such loc are met. By the definition of $\oplus$ and the induction hypothesis, consistency conditions in Definition 2.1 holds for other locations in the store.

Case **addm**, **replm**: If te_k is $\texttt{addm}\,(c,t,m,\overline{form},e)$, then by the rule of (ADD METHOD), we have $CT_1 \vdash \texttt{addm}\,(c,t,m,\overline{form},e), \gamma \rightsquigarrow \gamma, CT_2$. The rule does not affect γ and therefore does not affect the store (μ). Therefore, we have $\Pi' = \Pi$ and since Π is consistent, we conclude that Π' is consistent. Case **replm** is similar.

Case **addp**: If te_k is $\texttt{addp}\,(c,d)$, then for class c, operation te_k can be translated into a series of **addf** and **addm** operations to class c. Since both **addf** and **addm** are type safe, we conclude that **addp** is type safe and consistent with Π' ∎

In summary, our soundness result shows that provided the correct ordering of deployment expressions is enforced, and the semantics of deployment expressions shown in Figure 6 is maintained in the implementation, dynamic deployment of a well-typed ITD will maintain the type safety of the running code.

3. JVM Support for Dynamically Deployed Inter-type Declarations

Our formalization helps illustrate the key ideas, basic semantics, and requirements for supporting dynamic deployment of inter-type declarations in a virtual machine. We now turn to the implementation of these requirements in the JikesRVM [22]. Although the discussion in this section is geared towards JikesRVM, the presented modifications can be adapted to extend other JVMs. The requirements for VM support essentially boils down to the transformation rules defined in Figure 6.

The type checking rules for ITDs require minor adaptation to account for privacy modifiers in Java and exceptions. To account for exceptions in replacing a method we add a constraint that the new method's exception list is a subset of original method's exception list to avoid surprising the clients of the method. Interfaces are handled by type checking them as classes with no fields and empty method bodies and **super** is checked by consistent method renaming.

To deploy a safe sequence of transformation rules, we first stop the program, modify the class structures (VM equivalent of a class table) and then if necessary trigger a modified garbage collection phase that updates the objects as necessary to model modifications in the store. Since an ITD can introduce new fields, we need to visit each instance of modified classes. Considering that a typical garbage collector already performs this task, extending GC seems like the most natural and least intrusive way to modify JVM to support object layout update.

We have modified the `BaseBase` configuration of the JikesRVM and extended the SemiSpace and Generational MarkSweep collectors in JikesRVM to support instance updates. The `BaseBase` configuration does not optimize the running program but rather replaces bytecodes with equivalent machine code. Optimizations complicate dynamic method replacement since replaced methods might be inlined within other methods. Therefore, supporting dynamic replacement under optimization requires additional steps that greatly depend on the JVM implementation. We note that the implementation of Steamloom [6] uses the same JVM under the `FastAdaptive` configuration to support dynamic point-cut advices. This configuration emits optimized code when compiling methods. Therefore, several ideas might be adopted from Steamloom when porting our implementation to use the `FastAdaptive` configuration. However, we do not consider this to be a threat to the basic ideas discussed in this section. This section describes our implementation starting with a mapping from abstract terms in the semantics to concrete VM structures. Given a set of statically valid ITD declarations, and based on the transformations described by the operational semantics from the previous section, we will show how these transformations relate to JVM internals and how to extend the JVM to perform such transformations.

3.1 Mapping Class Table (CT) and Configuration (γ) to VM Structures

As expected of an operational semantics, our domain (shown in Figure 5) maps well to VM data structures. The fixed list of program's declaration CT maps to a special area in JikesRVM that stores the class information. Each class holds an array of members where each array element, points to the object representing a field or a method. The code for a method is stored within the virtual machine in two for-

mats: in bytecodes and in native machine code. Bytecodes are loaded from the class files. The Just-In-Time (JIT) compiler in the `BaseBase` configuration scans the method's bytecodes and translates them to native machine code. References to other methods and fields are encoded into the generated machine code. JIT finds the name of other methods and fields by consulting the class constant pool, which contains the mapping between entity names and their IDs used in the bytecodes. Our transformation rules modify the class constant pool.

3.2 Deployment Transformations in JikesVM

We start by discussing the technical challenges associated with reading bytecodes for the sequence of deployment expressions ($\overline{te}$). We then show the implementation of these deployment expressions, which modifies the constant pool and the heap.

Loading Deployment Expression Sequence

Our JikesRVM extension is implemented as a thread, which is initially idle. Upon users' request, the thread wakes up and searches for the deployment expressions to be applied. The thread takes bytecodes from the expressions compiled as self-contained class files (from here on called the *donating classes*) and integrates them into the classes of the running application (from here on called *receiving classes*). The modifier thread has two duties: (i) to change the class structures (e.g., method tables and constant pool) at runtime, and (ii) to change the bytecodes of the new methods.

As described previously, step (i) is needed for adding new methods in existing classes. Step (ii) is needed because the bytecodes of new methods are coming from donating classes that might have a different constant pool than the receiving classes. Using different constant pools can create syntactic and semantic errors. To ensure that the new bytecodes refer to the correct entities in the receiving classes, the modifier changes the entity IDs used by the method to match the IDs used in the receiving classes.

3.2.1 Implementing Deployment Expressions

The rest of the section describes how we supported our dynamic deployment expressions within JikesRVM. We show field addition (**`addf`**), method addition (**`addm`**), method replacement (**`replm`**) and parent addition (**`addp`**) expressions. The implementation of these is based on the rules described in Figure 6.

Field addition. The formal definition of this operation showed how the class table is affected and showed that the new field is propagated to sub-classes. Within the JVM, each class instance (i.e. object) contains values for the class's fields. By adding a field to the class definition, the object layout should be updated to accommodate the new field. To add a non-static field, the modifier (i) changes the class structure by adding the new field to the fields list in the class definition. (ii) propagates the field to sub-classes. (iii) Allocates heap space for the new field. (iv) recompiles methods that access the class's fields. The last two steps are specific to the VM's treatment of object record representation.

Steps (i) and (ii) are performed by extending the list of fields for the target class and its sub-classes. For step (iii), objects are manipulated to allocate the required space. To achieve this task, we relied on the JVM Garbage Collector (GC). This design decision is distinct from previous work on supporting changes in Java classes primarily because of a different emphasis on distribution of overhead. Malabarba *et al.* [31] support change in Java classes by extending the Java class loader functionality. However, their scheme requires monitoring accesses to the instances of modified classes, which has perpetual overhead, even after the class update. On the other hand, we favored a one-time overhead to deploy ITDs over a constant monitoring overhead after the deployment is finished.

To transform instances of modified classes, the updater thread forces a garbage collection after the addition of fields. We have extended GC to recognize objects of modified classes. When the garbage collector encounters an object instantiated from a class that has an added field, it increases the object size by the new field's size. However, increasing the object size requires copying it to another memory location. In this case, GC ensures that references to the moved objects are updated as part of its objects traversal. For the last step, since field offsets in an object might change after field addition, we need to recompile methods that access modified classes. Further discussion of this step appears as part of the implementation of the **`replm`** expression. Furthermore, since field additions are processed before method additions and replacements (Rule (T-ITD) in Figure 4), we can guarantee that no method will refer to non-existing field.

Handling static fields is straightforward. There is only one copy of such fields for each class stored by the JVM in a special region. Therefore, the modifier adds a static field by inserting a new entry in the list of static fields.

In our semantics, we only had reference types. In the full Java language however, fields can also have value types in addition to reference types (recently added `enum` types can be desugared to the value type **`final static int`**). We assign new fields their default values based on their types as defined by the Java language specification. It is the responsibility of the programmer to add code in new methods that accounts for this default value semantics. An alternate semantics is possible for value types, for which we could allow a default value to be specified. However, for reference types initializing the field may require running constructors from other classes, which may in turn call other methods. Adding this complexity to the deployment process did not seem to have a corresponding return, thus we omitted it. Furthermore, it is intuitive to see that the existing methods would not use these new fields, since they did not exist when these methods were written.

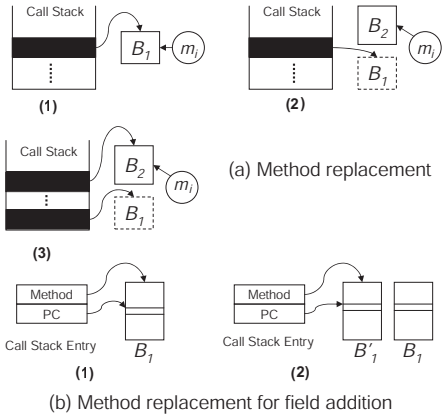

Figure 7. The effect of the deployment expression **`replm`** on the call stack. m_i is the method entry. B is the method body. PC is the program counter.

Method Addition. As the semantics of deployment expressions suggests, the bytecode for the new method are added to the method table of the receiving class. As noted before, one of the modifier thread tasks is to change the method bytecodes to accommodate the differences between the constant pools. Each reference to the constant pool in the new method is replaced with an equivalent reference in the receiving class constant pool. If no matching entry is found, the modifier creates the needed entry in the receiving class constant pool. Now the method can use the receiving class constant. Next, the modifier instructs the JIT compiler to translate the bytecodes into machine code. These steps are in-line with those defined formally in the **`addm`** rule. The key implementation challenge was in the translation of the constant pool.

Method Replacement. The semantics of **`replm`** states that new method lookups return the new method body, while active invocations (i.e. on stack) are not affected. Figure 7(a) shows a call stack before and after the operation. Initially, the method entry m_i points to the current method body B_1 (1). The new method implementation B_2 is linked to the method entry (2). Now, any call to that method will use the new version (3). The old code will remain in memory until all references to it from the call stack are removed (e.g. by `return` statements). The procedure described in Figure 7(a) allows for two versions of the method to exist.

For method replacements caused by field addition, the new and old method bodies have the same bytecode sequence. However, the new body has the correct field offsets within the object. Therefore, these methods have to be fully replaced even if they are on the call stack (Figure 7(b)). This is equivalent to α-renaming, which preserves type-safety. Initially, the call stack entry points to the current method body B_1 and the current location within the method (1). The modifier changes the call stack entry to point to the new implementation B'_1 (2). When the control is returned to the method, it will continue the execution using the new implementation. Field additions are done at GC time, at which the application methods are stopped. After GC, replaced methods will use the new objects and method bodies during the rest of its execution.

An alternate design would wait for a method to complete all its activations on the stack before replacing it. Such design would have the advantage that recursive invocation of a method will all use the same code, however, a disadvantage of such design is that for long running methods, the deployment may be delayed indefinitely which may defeat the purpose of investing in dynamic deployment. Moreover, a hybrid of these two strategies for method replacement is also possible, which selectively replaces a method based on annotations from programmers, but we haven't explored that in this paper.

Adding Parents. If a class d is added as a parent to class c, all of class d members should be inherited by class c. The **`addp`** operation is similar to a sequence of field/method additions. To complete this operation, our VM extension recognizes class d as a parent to class c and records this information in class c, d structures. This step is needed to ensure the correct operation of `cast` expressions. Like adding field and adding methods, correctness of this operation is also checked during static type-checking of inter-type declarations.

To summarize, we found that our implementation remained faithful to our semantics barring a few inert and statically checked changes, e.g. privacy modifiers, exceptional behavior, default value of value types, static fields, constant pool translation.

3.3 Choice of Garbage Collector

A major component in our scheme is supporting field addition by extending GC. Another possible choice would consist of updating objects at first access. The details for this approach is presented in Malabarba *et al.* [31]. However, monitoring object accesses has a high, long running, performance overhead, as each object access to modified types should be instrumented. Furthermore, once an object is updated and moved to a different location, references to this object should be updated reflecting its new location. In Malabarba *et al.* [31], this is accomplished by using object handles. For obvious reasons, object handles are avoided in modern JVMs. For these reasons, we have decided on extending GC to perform instance updates. As an added benefit, forced GC by our approach might negate the need for one regular GC down the road by eliminating dead objects.

Several GC algorithms exists, some of which are easier to extend than others. However, easier implementation does not necessary yield to better performance. When we started this work we extended the Semi-Space collector in JikesRVM. The semi-space collector divides the memory space into two segments. At each collection, live objects are copied from the "from" space to the "to" space. Only one space is used for allocations. This collector is easy to extend since all

objects are copied at GC time. Therefore, only minimum extension to copying logic is needed.

However, the semi-space collector has several disadvantages that prohibits its use in production-grade JVMs. The use of two spaces means that half of the memory is idle at any time. Therefore, applications need a heap twice as large as their live volume. Second, copying all live objects at runtime is expensive. Due to these limitations, other collection schemes are used such as generational collectors. Generational collectors are faster and consume less space than copying collectors. With generational collectors, objects residing in mature space are not moved. Therefore, supporting object update requires extending the collector with the ability to possibly move objects residing in the mature space along with necessary reference modification. In this paper we also present a modification to the generational mark-sweep collector in JikesRVM to enable field addition.

During a major collection in a generation collector, GC traverses objects in the heap and marks live ones in the mature space. After the mark phase, it sweeps the mature space and reclaim the space of dead objects. In our extension, when GC tries to mark an object that needs to be modified, it checks whether the new object layout fits in the same place as the old instance. The mature space in JikesRVM uses a freelist allocator that segregate objects based on their size class. Each size class is allocated from a separate list. This means that allocations proceed by an increment equal to the size class. As a result, some objects can have few free words equal to the difference between the object size and its size class. If the new fields size fit in the free space, we install the new object in place of the old one. Otherwise, a copy is needed since the object now belongs to a different size class. If the object is copied to a different location, we need to install a forwarding pointer in place of the old object. During the sweep phase, when GC examines a reference fields in an object, we check whether the reference field points to a forwarded object. In this case the forwarding pointer is read and stored in the reference field.

4. Case Study: Dynamically Deployed ITDs for Dynamic Updating

Armed with a virtual machine capable of dynamically deploying inter-type declarations, we set out to address what had been our goal all along, “to study the effectiveness of aspect-orientation in enabling dynamic deployment of features in long-running applications", where feature changes are expressed as aspects and applied dynamically as modular units of software update. This section describes our observations, which may we add, were very positive.

4.1 Background: Dynamic Updating Systems

During the life cycle of an application, many bugs are discovered that require immediate solution. Also, users might request new features as their requirements grow. Typically, supplying users with a new feature requires a service shutdown. For example, changes in daylight saving rules requires a JRE update and restart[1]. The disadvantage of a discontinued service ranges from user annoyance, to multi-million dollars of loss [36, 37].

For mission critical applications like transaction processing servers in business systems, being available 24/7 is equivalent to staying in business. A study found that 75% of nearly 6000 outages of high-availability applications were planned for hardware and software maintenance [30]. The causes of software updating include security patches, bug fixes or changes in policy. Solutions to maintain availability while updating requires redundancy (e.g. cluster rolling upgrade). However, such approaches are usually too expensive to be adopted by all users and slow down the upgrade process. Furthermore, software updating is not limited to large server applications. The increasing interest in adopting Java applications for mobile platforms (e.g. the Android platform by Google that runs on a JVM) where redundancy is absent presents a new challenge to application availability. Therefore, it is crucial to be able to update these applications without restarting them.

Due to the previous reasons, many researchers have investigated dynamic updating systems, where applications are updated amid their execution. Dynamic updating systems provide a low-interruption solution to the update problem. A more complete review of previously proposed systems is available in Section 6.

Our goal was to update Apache’s XML parser (Xerces) [50] library version from 1.2.3 to 1.3.1. Xerces is used to read, parse and validate XML documents. For this case study, we considered two release cycles of Xerces. Each cycle represents the difference between two consecutive versions and consists of a series of features added to implement the new version. The first cycle is between versions 1.2.3-1.3.0, which was 2 months long, the second between versions 1.3.0-1.3.1 and lasted for six weeks.

4.2 Expressing Xerces Changes Modularly

To that end, our first goal was to study the difference between the two versions of Xerces and represent these differences using pointcut-advice and/or inter-type declarations. The changes between the versions of Xerces are summarized in Figure 8. We note however, that we are not concerned with comparing ITD vs. pointcut-advice (PCA), but rather with showing why support for dynamic ITD can be beneficial for situations such as software updating.

Each row shows the changes needed to implement all features that constitute the release in a serialized manner. In other words, each feature is applied after the previous one. Therefore, the number shown represents the occurrence of changes summed for all features. For example, two changes to the same class are counted twice.

[1] http://java.sun.com/javase/timezones/index.html

Changes		1.2.3-1.3.0	1.3.0-1.3.1
Classes	Added	15	1
	Changed	82	56
Methods	Added	62	33
	Changed(PCA)	32	28
	Changed(ITD)	68	95
Fields	Add(Primitive)	28	18
	Add(Reference)	9	25
	Rename	7	5
Inner Classes	Added	9	1

Figure 8. Summary of features added to each version.

The row for fields shows the different types of added fields. The types are grouped into two categories: Primitive and Reference. The Primitive shows added fields of basic types besides reference. The reason for this distinction is that this type of fields can be easily introduced dynamically. However, for the second type (Reference), the initial value is unknown. Therefore, the developer needs to adapt the aspect to include special initialization code for these fields.

The changes between the versions of Xerces were analyzed and grouped into related changes that together implement a feature. This is similar to the work of Previtali and Gross [40, 41] who have proposed the use of aspects to model code evolution. They illustrated their hypothesis using the evolution data of a small client-server application and discussed possible implementation schemes. Our objective is to take the notion of software evolution even further to enable dynamic software updating using aspects.

Example	Changes			ITD
	Class	Method	Field	
Sec. 4.2.1	4	7	1	√
Sec. 4.2.2	1	2	1	√
Sec. 4.2.3	15	16	7	X
Sec. 4.2.4	4	10	0	√

Figure 9. Changes required to implement the example features presented. The table also shows whether these features can be implemented as an inter-type declaration. Non of these features can be implemented as a pointcut-advice.

We present several examples of added features that range from performance enhancement to new functionality. The changes required to implement each feature is presented in Figure 9. The figure shows the required modifications, in terms of number of classes, methods and fields changed or added, required to implement each of the four features. We also show whether the feature can be implemented as an inter-type declaration. In general, we found that using inter-type declarations allowed us to model more features than just pointcut-advice model. Although it is possible to emulate many ITD cases with certain PCA patterns such as delegation-based model [17, 45] or object-fragment mode, using ITD lends itself to better and easier design. Furthermore, such patterns can clearly affect long-running performance since it adds a level of indirection to added field accesses. Thus, it appears to be the case that software updating of such applications would benefit from our VM-based approach that supports dynamic deployment of inter-type declarations.

4.2.1 Feature: Optimizing XML Normalization

Normalization refers to the process by which whitespaces are removed from text nodes in XML documents. This process facilitates node comparisons and lookups. This feature was already implemented in Xerces 1.2.3, when normalization is requested, Xerces 1.2.3 traverses the XML node tree and normalize nodes. However, the traversal time can be reduced if nodes can indicate whether their children are normalized. In this case, the traversal process can skip whole sub-trees.

Adding this feature requires several changes to Xerces 1.2.3. By looking at the code of Xerces 1.3.0 (for which the performance enhancement was implemented), we extracted the following code changes: Class `AttrImpl`: two methods are added to check whether inserted nodes are normalized and whether nodes remain normalized after a removal. Class `AttrMap`: two methods are modified to check whether changed node's attributes are normalized. Class `ElementImpl`, the method `normalize` now checks whether the node is normalized, if not, it will traverse the sub-tree rooted at that node. Class `NodeImpl`: two methods and one field are added. These methods check whether a node is normalized, while the new field holds the unnormalized flag value.

This performance enhancement feature can be implemented as an inter-type declaration that will add necessary methods, fields and replace modified methods. The inter-type declaration adds the new methods in classes `AttrImpl,NodeImpl` and adds the new field to class `NodeImpl`. To fully describe these changes using aspects, an aspect containing an `around` advice will also be needed to replace the calls to the old methods with the modified version that support the feature. The new version of each method is implemented as an inter-type declaration method. The `around` advice directs calls to older version to added methods.

4.2.2 Feature: XML Notations

An XML notation element provides a mechanism to perform external validation on XML elements. The notation consists of several attributes that directs the XML validator to external validation resources. To support this functionality, the class `TraverseSchema` was modified to support notations. A new field is added (`fNotationRegistry`) to store notation elements. Also, the traversal method (`doTraverseSchema`) is modified to call a new method (`traverseNotationDecl`) when it encounters a notation element.

4.2.3 Feature: Identity Constraints

Certain items in an XML documents can carry special attributes that relate to their uniqueness or references throughout the document. These attributes are referred to as identity constraints. For example, the "unique" identity constraint is used to indicate that values of a certain element are unique (e.g. can be used as an index key in an XML-based database).

This feature was implemented by modifying 5 classes and adding a package containing ten classes that represent the constraints. All modified classes had fields and methods added to their implementation. Furthermore, an inner class was added to one of the modified classes. Due to the type of code changes, this feature can only be partially modeled using inter-type declarations. Pointcut-advice (PCA) model cannot handle method and field additions. While inter-type declarations can add these class members, the introduction of the inner class can not be expressed in our current model .

4.2.4 Feature: Build Time Optimization

When processing new documents, Xerces (as will other XML parsers) converts the document into a Deterministic Finite Automata (DFA). The DFA is constructed from supplied XML schema files and used to enhance the performance of queries over XML documents. A feature was added to Xerces that enhanced the speed of building the DFA by a factor of two. The feature affected four classes and required modifications to ten methods. For at least one class (`DFAContentModel`) the changes to methods were invasive and can not be represented by an `around`, `before` or `after` advice alone. Even if the advice body contained the new method implementation, advices can not call private methods or use private fields. Therefore, the changes required introducing new methods that has the modified implementation.

4.3 Characteristics of Update Aspects

Changes		1.2.3-1.3.0	1.3.0-1.3.1
Aspects	PCA	3	1
	ITD	15	22
	None	4	1
Features		22	24

Figure 10. Summary of features added to each version.

Figure 10 presents the number of ITD and PCA aspects needed to update Xerces. The breakdown of these aspects is presented in Figure 12. The first observation is that most changes can only be represented as inter-type declarations. Although the PCA model can be used to implement few features, the majority of features introduce changes that can not be modeled efficiently or cleanly as pointcut-advices only. We also note that some features can not be modeled using our model (or by using ITDs in AspectJ). These features require code changes that are unsupported by AspectJ such as introducing inner-classes that reference and are referenced by the owner class.

Tag	Classes		Method			Fields		Inner	Aspect		
	new	mod	new	PCA	ITD	safe	Ref	class	PCD	ITD	None
708	1	4	4							√	
710		1		1					√		
714		1			1					√	
716		1			1					√	
721		1			1					√	
722		1			1					√	
727		5	1		2	3				√	
729		10	3	1	6		2			√	
736		1		1					√		
742		1			1					√	
773		4			1					√	
776		9	18	2	7		2			√	
777		1	7	1				1			√
778		1			1					√	
779		1			5					√	
789	12	6	7	5	2		3	1			√
793		1	2		4					√	
796		9		10	8					√	
797	1	1	1	1	4			1			√
804		5	6	2	9					√	
817	1	18	13	8	15		2	6			√
829		2							√		
Avg.	0.68	3.73	2.82	1.45	3.09	0	0.41	0.41			

Figure 11. Summary of 22 features added (v.1.2.3-1.3.0.)

Tag	Classes		Method			Fields		Inner	Aspect		
	new	mod	new	PCA	ITD	safe	Ref	class	PCD	ITD	None
852		13	4	10	24		1			√	
853		4	11	1	1		2			√	
861	1	6	4	11	9		6			√	
872		1	1		4		5			√	
908		6	3	1	9		1			√	
914		2	4		4		1			√	
928		2			2					√	
930		1			1					√	
946		2			12					√	
949		2	1		3		1			√	
950		1			2	2	1			√	
951		1			1					√	
952		1		2			1			√	
953		3	3		4		2			√	
955		1			1					√	
956		1	2		8		4	1			√
957		1			1					√	
958		1			1					√	
962		1		1					√		
963		1		1	1					√	
967		1			1					√	
977		1			3					√	
980		2		2	1					√	
981		1			1					√	
Avg.	0.04	2.29	1.38	1.13	3.96	0.08	1.04	0.04			

Figure 12. Summary of 24 features added (v.1.3.0-1.3.1.)

5. Performance Evaluation

As mentioned previously, a pragmatic dynamic updating solution must have a near negligible steady-state overhead and acceptable overhead during the updating process. This section presents the results of a performance evaluation, which analyzes our VM extension for these desirable properties.

All experiments shown were conducted on a machine with AMD Athlon X2 Dual Core 3.0GHz processor and 2GB RAM. Results reported are the average of 11 runs after excluding the maximum and minimum readings.

5.1 Steady-State Overhead

In this experiment, we measured the steady-state overhead of using of our extended Java virtual machine compared to the unmodified JikesRVM. This experiment reflects the cost of having the option to dynamically update long-running applications. We executed several applications using the extended JVM and using the unmodified JVM, then compared their execution time. The extended JVM was not presented with a dynamic deployment sequence ($\overline{te}$). Therefore, the difference in execution time reflects the overhead of supporting dynamic deployment of inter-type declarations. For this experiment, we used SPECJVM98 and Dacapo [4] benchmarks, which represents a variety of Java applications. These results are shown in Table 13.

Application	Exec time	Application	Exec time
compress	1.01	eclipse	1.01
jess	1.0	fop	1.03
db	1.02	Hsqldb	1.05
javac	1.01	jython	1.06
mpegaudio	1.01	luindex	1.02
Mtrt	1.04	lusearch	1.05
jack	1.03	PMD	1.06
Geometric Mean	1.016		1.06

Figure 13. Overhead in extended JVM

Figure 13 shows the normalized execution time for the extended JVM compared to the unmodified JVM. Results higher than one means that the extended JVM is slower than the unmodified JVM. In general, our platform has low overhead compared to the unmodified JVM. Furthermore, since the same VM code is executed before and after an update (e.g. no indirection), the results also represent the after-update steady state overhead.

5.2 Deployment Overhead

Our extended JVM shows a low steady-state overhead for dynamically deploy ITDs. However, beside steady-state overhead, we are also concerned with deployment time and its effect on performance. We will show that our approach has acceptable performance degradation for the short duration of the deployment process. The period of time required to apply changes is directly related to the number and type of required modifications. For example, adding fields require more time than adding methods since a garbage collection is required. We present three experiments to measure the dynamic weaving overhead. First, we selected few features from the Xerces case study and measured the time required to apply each feature. Second, since adding fields to a class is associated with a garbage collection, we present an experiment using a synthetic benchmark to evaluate the effect of adding fields on GC performance. Finally, we measured the performance before, while and after performing a dynamic deployment. This experiment helps in evaluating performance degradation during deployment.

5.2.1 Dynamic Deployment Overhead.

In this experiment we measured the dynamic deployment time and studied the different phases of the deployment process. We selected seven aspects representing features/bug fixes that were added to Xerces V1.2.3.

Figure 14 shows the duration of each deployment phase. The table shows time consumed in loading (reading bytecodes) from file, compiling (compiling bytecodes into machine code) and applying the aspect (changing class structure). We note that deployment time is dominated by loading the bytecodes of new and modified methods. As for compilation time, it is affected by the type and size of added bytecodes. Lastly, updating the class structure has negligible time compared to the other phases in the weaving process.

Tag	Load (ms)	Compile (ms)	Apply (ms)
708	131.18	6.09	2.55
710	81.09	1.91	0.18
714	34.09	4.91	0.18
716	81.73	2.45	0.09
721	31.55	0.18	0.82
722	32	0.73	0.18
727	159.09	18.45	1.82

Figure 14. Dynamic Deployment Performance

5.2.2 Garbage Collection Performance.

In the previous experiment, the amount of work performed by GC depends on the number of live objects when the update is applied. To better understand the overhead of instance updating through GC, this section presents an experiments to evaluate the performance of our instance updating approach.

Adding fields to a class requires modifying all of the class instances. We carry this transformation by extending the functionality of garbage collection. After modifying the class structure to recognize the new field, GC is invoked to complete the update and to inspect and transform all of the class's instances.

In this experiment we measure the effect of object transformation on GC time. Our target application is a synthetic benchmark where we can control the number of objects affected by field addition. The benchmark consists of a payload class composed of three integer fields and a reference field. The payload class instances are connected in a linked list using the reference field. The application allocates a predefined number of payload objects and connects them. Then, the application traverses the list and touches all elements.

We compared the GC time between three scenarios. In all scenarios GC copies objects from one space to another. The difference in these scenarios is in the size of copied objects. In the first scenario, the GC runs normally (i.e. base case). It copies the object without modification and thus the size remains the same (4 words). The second scenario represents the additional cost needed to update instances. Objects are extended to accommodate an additional integer field. Therefore, the object size is increased by one word

(4 -> 5). The final scenario represents the GC operation after the update. The GC in the third scenario will copy the larger objects (5 -> 5) without inspection. The results in Figure 15 were obtained using the SemiSpace collector. We note that the Generational MarkSweep collector showed similar results for the instance update case (second scenario). The object sizes of four and five words belong to different size classes. Therefore, the Generational collector needs to copy the whole object similar to the SemiSpace collector.

# of Objects	GC time (s)		
	4->4	4->5	5->5
10K	0.29	0.35	0.3
100K	0.49	0.78	0.5
1M	2.77	4.66	2.77
10M	23.00	46.28	22.99
11M	25.37	51.26	25.42
12M	27.52	55.59	27.56
13M	29.79	59.80	29.81

Figure 15. Garbage Collection Performance

Figure 15 shows the GC timing for the three scenarios. We ran seven experiments and varied the number of instantiated payload objects. The maximum number of objects in this experiment (13M) is higher than what is expected in regular applications. For example, the largest live volume from Dacapo benchmarks is 72MB (hsqldb) corresponding to about 3.2M objects. The use of large number of objects in this experiment helps in understanding the overhead of object modification in relation to copying costs.

We note that with higher number of modified objects, the cost of adding fields dominates and reduces GC efficiency. GC time is affected due to two factors. First, the additional cost of updating objects. Second, due to additional requests for additional memory space. Note that the first and third scenarios have similar performance although the object size is different. This indicates that the extra time in the second scenario is not attributed directly with the increase in object size, but rather to the additional checks and space requests.

5.2.3 Relative Performance.

The last experiments investigates whether applying features dynamically can affect their performance. This experiment helps in evaluating performance degradation during the weaving process. For this experiment, we measured how many files a test code can process per second. Figure 16 shows the performance when we dynamically upgraded Xerces to tag 708. This update adds a feature (CR schema support) to Xerces. The figure shows the drop in performance during deployment. The deployment duration includes the time needed for GC. We also note that there is no drop in performance after the upgrade.

6. Related Work

The closest related work is by Malbarba *et al.* [31]. The similarity is that this work also allows runtime updating of object instances in a long-running application. However, there

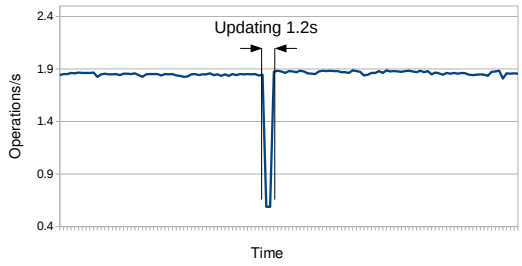

Figure 16. Performance before, while and after deployment

are several differences between techniques used by this work (and other work that are similar in spirit) and our approach. First, this method of object instance update relies on object handle in JDK 1.2, so they only need to update handles, which makes updates easier [31, Section-3.1,pp-349]. However, supporting object handles in modern JVMs can have severe performance penalties so neither JikesRVM nor modern Hotspot supports these. Second, this method of instance updating requires access instrumentation and locking [31, Section-3.2,pp-352]. Object access instrumentation can have prohibitive long-running overhead. On the other hand, our approach avoids long-running overhead by forcing GC at update time. The forced GC has the additional advantage of eliminating dead objects in heap and delay future GC in long-running application.

In the rest of this section, we discuss other closely related ideas: run-time weaving, load-time weaving, VM support for dynamic adaptation, and online updating.

Run- and Load-Time Weaving. There are several approaches for runtime weaving such as PROSE [39], HandiWrap [3], Eos [43], etc. A typical approach to runtime weaving is to attach hooks at all join points in the program at compile-time. The aspects can then use these hooks to attach and detach advice at run-time. A load-time weaving approach delays weaving of crosscutting concerns until the class loader loads the class file and defines it to the VM [29]. Load-time weaving approaches typically provide weaving information in the form of XML directives or annotations. The aspect weaver then revises the assemblies or classes according to weaving directives at load-time. A custom class loader is often needed for this approach. There are load-time weaving approaches for both Java and the .NET framework. For example, AspectJ [23] has load-time weaving support. Weave.NET [28] uses a similar approach for the .NET framework. The JMangler framework can also be used for load-time weaving [26]. It provides mechanisms to plug-in class-loaders into the JVM. However, neither runtime nor load-time weaving approaches support dynamic deployment of inter-type declarations, where the primary challenge is in efficient runtime updating of instances.

Virtual-Machine Support for Dynamic Weaving. Adaptation of a running object-oriented program has been approached from different directions by several projects in

past. We are aware of three projects that have sought to support aspects at the virtual machine level. *Steamloom* [6], *PROSE2* [38], and *Nu*[12]. Our work is similar to Steamloom in that it also extends the Jikes Research VM, an open source Java VM [22], however, it also supports dynamically deployed inter-type declarations. On the other hand, PROSE2 proposes an enhanced implementation for the original PROSE approach, by incorporating an execution monitor for join points into the virtual machine. This execution monitor is responsible for notifying the AOP engine which in turn executes the corresponding advice. This model is suitable for implementing pointcut-advice model; however, it cannot be used to model dynamically deployed inter-type declarations. Golbeck et al. propose lightweight support in virtual machines for AspectJ [15], however, they do not support dynamic deployment of inter-type declarations, whereas we do. Our own previous work on the *Nu* virtual machine [12] is also limited in that sense. It only supports the pointcut-advice model [12].

Haupt and Schippers's delegation-based model [17, 45] and Kuhn and Nierstrasz's object-fragment model [27] are capable of expressing dynamic deployment of inter-type declarations, however, efficient implementation of neither approaches has been demonstrated as of this writing. Our work could be seen as a potential stepping stone for demonstrating efficient implementations of these models.

Our work is also related to ideas that allow object representations to be adapted at runtime such as Nielsen and Ernst's proposal on VM support for virtual classes [35] in the gbeta language [13] and similar support in CeasarJ [33] and ObjectTeams [18]. In these work runtime adaptation of objects and classes is often part of the language semantics in the form of mixin-like composition. Our work is similar to CeasarJ [33] and ObjectTeams [18] in that we also aim to provide support in a Java language environment although unlike these approaches, our work tackles use cases where a class definition may not be computed statically (primarily because the rest of the class definition may not have been conceived at the time of compilation). The main difference from the gbeta VM is due to the host VMs.

Online Update. Our case study in similar in spirit to many of the dynamic updating approaches such as Ginseng [34] and POLUS [9] that provide facilities for dynamic software updating for C programs. Malabarba *et al.* [31], Kim *et al.* [24, 25] and Subramanian *et al.* [47] provide similar facility for managed languages. However, unlike many of these approaches that express upgrades in the form of a changed code, class, or methods our main emphasis in the case study was on enabling a *modular units of software upgrade* in the form of aspects that describe only the changes made due to an upgrade and a declarative representation of where the change ought to be applied. Moreover, the mechanisms that we have developed are not limited to dynamic software updating only, rather updating is presented as a use case for a more general support for dynamic deployment of ITDs.

Similar to Hicks *et al.* [19] and Stoyle *et al.* [46]'s work we have also developed conditions for safe deployment of an ITD. The main difference is that Hicks *et al.* [19] and Stoyle *et al.* [46]'s work is in the context of the C language, whereas our work is in the context of an OO calculus.

A large body of work related to dynamic updating exists under SmallTalk VMs. Although they share many of the updating constructs presented here and in other dynamic updating literature, SmallTalk-based updating is program-guided. Updates are specified and controlled by the running program. In our case, the updates are controlled by the VM. Using the VM to control the update process presents the challenges of correctly ordering and loading external updates.

7. Conclusion and Future Work

Dynamic aspect deployment is an important feature of an aspect-oriented language design that has many applications, such as runtime monitoring [7], runtime adaptation to fix bugs or add features to long running applications [48], runtime update of dynamic policy changes [38]. Current dynamic aspect weaving schemes support pointcut-advice model of Masuhara and Kiczales. In this work, we described the design and implementation for efficient support of dynamic deployment of inter-type declarations.

Several schemes have been proposed to support dynamic deployment of aspects. *Steamloom* [6], *PROSE2* [38] and *Nu* [12] present an aspect-aware Java VM. Steamloom moves weaving into the VM, which allows preserving the original structure of the code after compilation and shows performance improvements of 2.4 to 4 times when compared to AspectJ. PROSE2 incorporates an execution monitor for join points into the virtual machine. Nu presents an efficient JVM support for pointcut-advice aspects by extending Sun's HotSpot JVM. These schemes support dynamic weaving of pointcut-advice. However, we have seen that pointcut-advice aspects can not be used to express certain type of code features. For example, only ITDs can be used to add fields or methods to classes.

In this work, we described the design and implementation of our strategy for efficiently supporting dynamic deployment of inter-type declarations in JVM. We presented the usefulness of supporting dynamic ITD through a detailed real world case study. The case study demonstrated how ITD can be used to describe software features. Therefore, supporting dynamic deployment of inter-type declaration serves as an approach for online update of running applications.

Our scheme presents several advantages over dynamic pointcut-advice weaving schemes. By extending the JVM, we are able to support not only pointcut-advice aspects, but ITD aspects as well. Our implementation shows negligible overhead before deploying aspects and can apply aspects in relatively short time.

Future work includes extending our implementation in several directions. First, we will examine dynamic deployment mechanism when optimizing compilers are used within the JVM. This will facilitate the task of supporting dynamic aspects without sacrificing performance. Another direction where we envision future work is to study dynamic weaving to active methods. Our current implementation does not support replacing active method bodies.

References

[1] C. Anderson and S. Drossopoulou. δ: an imperative object based calculus with delegation. In *USE*, 2002.

[2] P. Avgustinov, J. Tibble, and O. de Moor. Making trace monitors feasible. In *OOPSLA '07*, pages 589–608, 2007.

[3] J. Baker and W. Hsieh. Runtime aspect weaving through metaprogramming. In *AOSD '02*, pages 86–95. ACM, 2002.

[4] Blackburn, S. *et al.*. The dacapo benchmarks: java benchmarking development and analysis. In *OOPSLA '06*, pages 169–190, 2006.

[5] C. Bockisch, M. Arnold, T. Dinkelaker, and M. Mezini. Adapting virtual machine techniques for seamless aspect support. In *OOPSLA '06*, pages 109–124. ACM, 2006.

[6] C. Bockisch, M. Haupt, M. Mezini, and K. Ostermann. Virtual machine support for dynamic join points. In *AOSD '04*, pages 83–92. ACM, 2004.

[7] E. Bodden, F. Chen, and G. Rosu. Dependent advice: a general approach to optimizing history-based aspects. In *AOSD*, pages 3–14, 2009.

[8] C. Allan *et al.*. Adding trace matching with free variables to AspectJ. In *OOPSLA '05*, pages 345–364. ACM, 2005.

[9] H. Chen, J. Yu, R. Chen, B. Zang, and P.-C. Yew. POLUS: A POwerful Live Updating System. In *ICSE*, 2007.

[10] C. Clifton. A design discipline and language features for modular reasoning in aspect-oriented programs. Technical Report 05-15, Iowa State University, Jul 2005.

[11] R. Dyer and H. Rajan. Nu: a dynamic aspect-oriented intermediate language model and virtual machine for flexible runtime adaptation. In *AOSD*, 2008.

[12] R. Dyer and H. Rajan. Supporting dynamic aspect-oriented features. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2009.

[13] E. Ernst. *gBeta—A Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritiance*. PhD thesis, University of Aarhus, Denmark, 1999.

[14] M. Flatt, S. Krishnamurthi, and M. Felleisen. A programmer's reduction semantics for classes and mixins. In *Formal Syntax and Semantics of Java*, chapter 7, pages 241–269. 1999.

[15] R. M. Golbeck, S. Davis, I. Naseer, I. Ostrovsky, and G. Kiczales. Lightweight virtual machine support for AspectJ. In *AOSD '08*, pages 180–190. ACM, 2008.

[16] J. Gosling, B. Joy, and G. L. Steele. *The Java Language Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[17] M. Haupt and H. Schippers. A machine model for aspect-oriented programming. In *ECOOP '07*, pages 501–524. Springer, 2007.

[18] S. Herrmann. Object teams: Improving modularity for crosscutting collaborations. In *NET OBJECT DAYS*, pages 248–264. Springer, 2002.

[19] M. Hicks. Dynamic software updating. *PhD thesis, Dept. of Computer Science, University of Pennsylvania*, 2001.

[20] R. Hirschfeld and S. Hanenberg. Open aspects. *Computer Languages, Systems & Structures*, 32(2-3):87–108, 2006.

[21] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *OOPSLA*, pages 132–146, 1999.

[22] JikesRVM Reasearch Virtual Machine. http://www.jikesrvm.org/.

[23] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *ECOOP '01*, pages 327–353. Springer, June 2001.

[24] D. Kim, Y. Jiao, and E. Tilevich. Flexible and efficient in-vivo enhancement for grid applications. *International Symposium on Cluster Computing and the Grid*, 2009.

[25] D. Kim, M. Song, E. Tilevich, C. Ribbens, and S. Bohner. Dynamic software updates for accelerating scientific discovery. *Proceedings of the International Conference on Computational Science*, 2009.

[26] G. Kniesel, P. Costanza, and M. Austermann. Jmangler-a framework for load-time transformation of Java class files. In *SCAM '01*, pages 100–110. IEEE CS, 2001.

[27] A. Kuhn and O. Nierstrasz. Composing new abstractions from object fragments. In H. Rajan, editor, *VMIL*, 2008.

[28] D. Lafferty and V. Cahill. Language-independent aspect-oriented programming. In *OOPSLA '03*, 2003.

[29] S. Liang and G. Bracha. Dynamic class loading in the Java virtual machine. In *OOPSLA '98*, pages 36–44. ACM, 1998.

[30] D. E. Lowell, Y. Saito, and E. J. Samberg. Devirtualizable virtual machines enabling general, single-node, online maintenance. *ASPLOS*, 2004.

[31] S. Malabarba, R. Pandey, J. Gragg, E. Barr, and J. F. Barnes. Runtime support for type-safe dynamic java classes. In *ECOOP '00*, pages 337–361.

[32] H. Masuhara and G. Kiczales. Modeling crosscutting in aspect-oriented mechanisms. In *ECOOP*, 2003.

[33] M. Mezini and K. Ostermann. Conquering aspects with caesar. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 90–99, New York, NY, USA, 2003. ACM Press.

[34] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol. Practical dynamic software updating for c. *PLDI*, 2006.

[35] A. B. Nielsen and E. Ernst. Virtual class support at the virtual machine level. In *3rd workshop on Virtual Machines and Intermediate Languages*, Oct 2009.

[36] D. L. Oppenheimer *et al.*. Roc-1: Hardware support for recovery-oriented computing. *IEEE Trans. Computers*, 2002.

[37] S. Parker. A simple equation: It on = business on. Technical report, Hewlett Packard, 2001.

[38] A. Popovici, G. Alonso, and T. Gross. Just-in-time aspects: efficient dynamic weaving for Java. In *AOSD '03*, pages 100–109.

[39] A. Popovici, T. Gross, and G. Alonso. Dynamic weaving for aspect-oriented programming. In *AOSD '02*, pages 141–147.

[40] S. Previtali and T. Gross. Dynamic updating of software systems based on aspects. In *ICSM*, pages 83–92, 2006.

[41] S. Previtali and T. Gross. Extracting updating aspects from version differences. In *LATE*, pages 1–5, 2008.

[42] H. Rajan and G. T. Leavens. Ptolemy: A language with quantified, typed events. In *ECOOP*, pages 155–179, 2008.

[43] H. Rajan and K. J. Sullivan. Classpects: unifying aspect- and object-oriented language design. In *ICSE '05*, pages 59–68.

[44] H. Rajan and K. J. Sullivan. Eos: instance-level aspects for integrated system design. In *ESEC/FSE-11*, pages 297–306, Sept 2003.

[45] H. Schippers, D. Janssens, M. Haupt, and R. Hirschfeld. Delegation-based semantics for modularizing crosscutting concerns. In *OOPSLA*, pages 525–542, 2008.

[46] G. Stoyle, M. Hicks, G. Bierman, P. Sewell, and I. Neamtiu. Mutatis Mutandis: Safe and flexible dynamic software updating. *ACM ToPLAS*, 2006.

[47] S. Subramanian, M. Hicks, and K. S. McKinley. Dynamic software updates: a vm-centric approach. *SIGPLAN Not.*, 44(6):1–12, 2009.

[48] D. Suvée, W. Vanderperren, and V. Jonckers. Jasco: an aspect-oriented approach tailored for component based software development. In *AOSD '03*, pages 21–29.

[49] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, Nov 1994.

[50] Xerces XML Parser Library from Apache Foundation. http://xerces.apache.org/xerces-j/.

A. Omitted Details of the Object-oriented Core Calculus

This section gives the small step semantics and the type system for the object-oriented calculus used in Section 2.1. Following earlier work [49], we define the calculus as a set of evaluation contexts $\mathbb{E}$ (shown below) and a one-step reduction relation that acts on the position in the overall expression identified by the evaluation context.

$$\begin{array}{rcl}\mathbb{E} & ::= & - \mid \mathbb{E}\,.\,m(e\ldots) \mid v\,.\,m(v\ldots\mathbb{E}\,e\ldots) \\ & \mid & \texttt{cast}\ t\,\mathbb{E} \mid \mathbb{E}\,.\,f \mid \mathbb{E}\,.\,f{=}e \\ & \mid & v\,.\,f{=}\mathbb{E} \mid t\ var{=}\mathbb{E};\ e \mid \mathbb{E};\ e\end{array}$$

This avoids the need for writing out standard congruence rules and clearly presents the order of evaluation. The language uses a strict leftmost, innermost evaluation policy, which uses call-by-value. The rules for standard OO expressions are given in Figure 17. We omit the treatment for standard exceptional conditions. In the semantics, the auxiliary method *lookup* (not shown) is used to look up the body of the method.

Evaluation relation: $\hookrightarrow : \Gamma \to \Gamma$

(NEW)
$$\frac{loc \notin dom(\mu) \qquad \mu' = \mu \oplus \{loc \mapsto [c\,.\,\{f \mapsto \texttt{null} \mid f \in fieldsOf(c)\}]\}}{\langle \mathbb{E}[\texttt{new}\ c()], \mu\rangle \hookrightarrow \langle \mathbb{E}[loc], \mu'\rangle}$$

(SEQUENCE)
$$\frac{}{\langle \mathbb{E}[v; e], \mu\rangle \hookrightarrow \langle \mathbb{E}[e], \mu\rangle}$$

(CALL)
$$\frac{loc \in dom(\mu) \qquad [c\,.\,F] = \mu(loc) \qquad (c', m(t_1\ var_1, \ldots, t_n\ var_n)\{e\}) = lookup(c, m) \qquad c <: c' \qquad e' = e[v_1/var_1, \ldots, v_n/var_n]}{\langle \mathbb{E}[loc.m(v_1, \ldots, v_n)], \mu\rangle \hookrightarrow \langle \mathbb{E}[e'], \mu\rangle}$$

(SET)
$$\frac{loc \in dom(\mu) \qquad [c.F] = \mu(loc) \qquad \mu' = \mu \oplus \{loc \mapsto [c.F \oplus (f \mapsto v)]\}}{\langle \mathbb{E}[loc.f = v], \mu\rangle \hookrightarrow \langle \mathbb{E}[v], \mu'\rangle}$$

(CAST)
$$\frac{loc \in dom(\mu) \qquad [c'\,.\,F] = \mu(loc) \qquad c' <: c}{\langle \mathbb{E}[\texttt{cast}\ c\ loc], \mu\rangle \hookrightarrow \langle \mathbb{E}[loc], \mu\rangle}$$

(GET)
$$\frac{loc \in dom(\mu) \qquad [c.F] = \mu(loc) \qquad v = F(f)}{\langle \mathbb{E}[loc.f], \mu\rangle \hookrightarrow \langle \mathbb{E}[v], \mu\rangle}$$

Figure 17. Semantics, based on [10, 42]. (Uses the overriding operator $\oplus$ for μ and σ in some rules).

The (NEW) rule says that the store is updated to map a fresh location to an object of the given class that has each of its fields set to null. These rules use $\oplus$ as an overriding operator for finite functions. That is, if $\mu' = \mu \oplus (loc \mapsto v)$, then $\mu'(loc') = v$ if $loc' = loc$ and otherwise $\mu'(loc') = \mu(loc')$. The *fieldsOf* function uses the class table (CT) to determine the list of field declarations for a given class (and its superclasses), considered as a mapping from field names to their types. Other rules are also standard.

(T-NEW)
$$\frac{isClass(c)}{\Pi \vdash \texttt{new}\ c() : c}$$

(T-LOC)
$$\frac{\Pi(loc) = t}{\Pi \vdash loc : t}$$

(T-GET)
$$\frac{\Pi \vdash e : c \qquad fieldsOf(c)(f) = t}{\Pi \vdash e.f : t}$$

(T-VAR)
$$\frac{\Pi(var) = t}{\Pi \vdash var : t}$$

(T-NULL)
$$\frac{isClass(c)}{\Pi \vdash \texttt{null} : c}$$

(T-CAST)
$$\frac{isType(t) \qquad \Pi \vdash e : u}{\Pi \vdash \texttt{cast}\ t\ e : t}$$

(T-SEQUENCE)
$$\frac{\Pi \vdash e_1 : t_1 \qquad \Pi \vdash e_2 : t_2}{\Pi \vdash e_1; e_2 : t_2}$$

(T-SET)
$$\frac{\Pi \vdash e : c \qquad fieldsOf(c)(f) = t \qquad \Pi \vdash e' : t' \qquad t' <: t}{\Pi \vdash e.f = e' : t'}$$

(T-CALL)
$$\frac{\forall\, i \in \{1..n\}\,.\,\Pi \vdash e_i : u_i \qquad \forall\, i \in \{1..n\}\,.\,u_i <: t_i \qquad \Pi \vdash e : c \qquad (c', t\ m\ (t_1\ var_1, \ldots, t_n\ var_n)\ \{e\}) = lookup(c, m)}{\Pi \vdash e.m(e_1, \ldots, e_n) : t}$$

Figure 18. Type-checking rules for expressions based on [10, 42].

The type checking rules for OO expressions are shown in Figures 18. The rules for top-level declarations such as class, methods, fields, are fairly standard (thus omitted).