Centrum voor Wiskunde en Informatica

**REPORT**RAPPORT

SEN

Software Engineering

*Software ENgineering*

# Semantics-Directed Implementation of Method-Call Interception

Ralf Lämmel, Christian Stenzel

**REPORT SEN-E0328 DECEMBER 23, 2003**

CWI is the National Research Institute for Mathematics and Computer Science. It is sponsored by the Netherlands Organization for Scientific Research (NWO).
CWI is a founding member of ERCIM, the European Research Consortium for Informatics and Mathematics.

CWI's research has a theme-oriented structure and is grouped into four clusters. Listed below are the names of the clusters and in parentheses their acronyms.

Probability, Networks and Algorithms (PNA)

Software Engineering (SEN)

Modelling, Analysis and Simulation (MAS)

Information Systems (INS)

Copyright © 2003, Stichting Centrum voor Wiskunde en Informatica
P.O. Box 94079, 1090 GB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

ISSN 1386-369X

# Semantics-Directed Implementation of Method-Call Interception

ABSTRACT

We describe a form of method-call interception (MCI) that allows the programmer to superimpose extra functionality onto method calls at run-time. We provide a reference semantics and a reference implementation for corresponding language constructs. The setup applies to class-based, statically typed, compiled languages such as Java. The semantics of MCI is used to direct a language implementation with a number of valuable properties: simplicity of the implementational model and run-time adaptation capabilities and static type safety and separate compilation and reasonable performance. Our implementational development employs sourcecode instrumentation. We start from a naive implementational model, which is subsequently refined to optimise program execution. The implementation is assessed via benchmarks.

*1998 ACM Computing Classification System:* D.3.3

*Keywords and Phrases:* Method-Call Interception; System Adaptation; Unanticipated Software Evolution; Aspect-Oriented Programming; Dynamic Weaving; Semantics-Directed Language Implementation; Source-Code Instrumentation; Benchmarking

# Semantics-Directed Implementation of Method-Call Interception

Ralf Lämmel [1,2] and Christian Stenzel [3]
ralf@cs.vu.nl and stenzel@informatik.uni-kl.de

[1] Vrije Universiteit, Amsterdam, The Netherlands
[2] CWI, Amsterdam, The Netherlands
[3] Technische Universität Kaiserslautern, Germany

**Abstract**

We describe a form of method-call interception (MCI) that allows the programmer to superimpose extra functionality onto method calls at run-time. We provide a reference semantics and a reference implementation for corresponding language constructs. The setup applies to class-based, statically typed, compiled languages such as Java. The semantics of MCI is used to direct a language implementation with a number of valuable properties: simplicity of the implementational model *and* run-time adaptation capabilities *and* static type safety *and* separate compilation *and* reasonable performance. Our implementational development employs source-code instrumentation. We start from a naive implementational model, which is subsequently refined to optimise program execution. The implementation is assessed via benchmarks.

**Keywords**: Method-Call Interception, System Adaptation, Unanticipated Software Evolution, Aspect-Oriented Programming, Dynamic Weaving, Semantics-Directed Language Implementation, Source-Code Instrumentation, Benchmarking

# Contents


**1 Introduction** **3**

**2 Semantics of MCI** **8**
2.1 Dynamic semantics of MCI constructs . . . . . . . . . . . . . . . . . . 9
2.2 Dynamic semantics of MCI-enabled method calls . . . . . . . . . . . . 11
2.3 Dynamic semantics of superimposed functionality . . . . . . . . . . . . 13
2.4 Static semantics of MCI constructs . . . . . . . . . . . . . . . . . . . . 15

**3 Naive implementation of MCI** **18**
3.1 The MCI registry . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
3.2 Instrumentation of method calls . . . . . . . . . . . . . . . . . . . . . 20
3.3 Compilation of superimpositions . . . . . . . . . . . . . . . . . . . . . 23

**4 Refinements of the MCI implementation** **25**
4.1 Caching MCI inquiries . . . . . . . . . . . . . . . . . . . . . . . . . . 25
4.2 Bypassing MCI inquiries . . . . . . . . . . . . . . . . . . . . . . . . . 27
4.3 Class-based interception . . . . . . . . . . . . . . . . . . . . . . . . . 29
4.4 Lazy MCI inquiries . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
4.5 Method wrappers . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32

**5 Benchmarks for the MCI implementation** **32**
5.1 Prime MCI model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
5.2 Comparison with static weaving technology . . . . . . . . . . . . . . . 38
5.3 Comparison with dynamic weaving technology . . . . . . . . . . . . . 39
5.4 Size of instrumented code . . . . . . . . . . . . . . . . . . . . . . . . . 40
5.5 Method call chart . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40

**6 Related work** **41**

**7 Concluding remarks** **43**

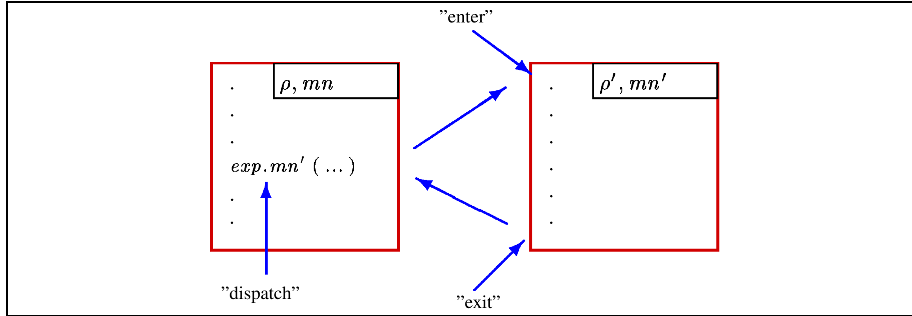



Figure 1: Points along the execution of a method call

# 1 Introduction

**Method-call interception** System adaptation means to modify the structure or the behaviour of a software system. System adaptation can happen at compile time, when the system is 'offline', or at run-time. In the present paper, we focus on method-call interception (MCI) — by what we mean run-time adaptation for object-oriented software systems such that method calls are subject to adaptation. That is, one *superimposes* additional functionality onto method calls while the program is up and running. The sketch of an example is this: "For a given period of time, log all instances of class XYZ as they occur as arguments of method calls to instances of class ABC." Functionality can be superimposed onto different points along the execution of a method call. In Figure 1, we illustrate some principle points; be it in a language like Java. The shown call departs from a method *mn* of an object $\rho$, and it invokes a method $mn'$ of an object $\rho'$. There are distinguished points *dispatch*, *enter* and *exit*. The *dispatch* point is placed right after the evaluation of the callee. The *enter* point is placed after the additional evaluation of the method arguments, that is, when we are about to *enter* the body of the called method. The *exit* point is placed after the execution of the method body, that is, when we are about to *exit* the called method. This form of method-call interception was initiated in [34] with focus on its semantical development. In the present paper, we focus on semantics-directed implementation of MCI.

**Run-time evolution** System adaptation with MCI can ultimately be used as means for (run-time) system evolution. There are two dimensions of evolution:

- *The decoration of method calls with functionality*: Superimposed functionality can be viewed as decorators $D_1$, $D_2$, ... that are added to a method $M$ one-by-one. We might also deactivate, reactive, and remove decorators. We might as well revise a decorator. These scenarios are illustrated in Fig. 2.
- *The quantification of relevant method calls*: We might want to include additional calls or exclude so-far intercepted calls – maybe by relaxing or restricting patterns for caller and callee sites. We use the term *event* for specifications of relevant method calls. An evolving event is illustrated in Fig. 3.

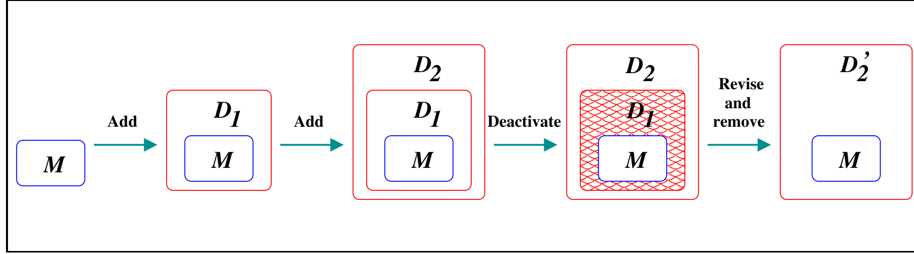



Figure 2: Run-time evolution of decoration

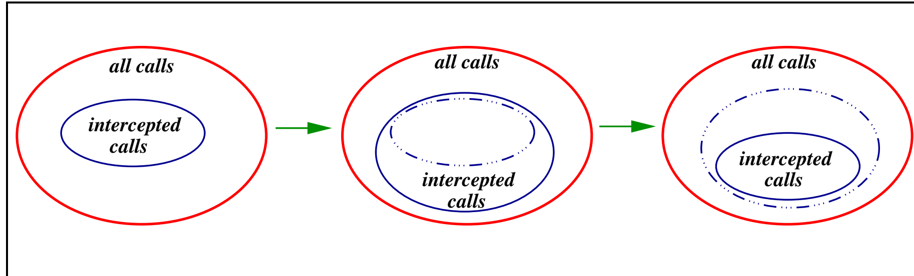



Figure 3: Run-time evolution of quantification

In the present paper, we focus on *adding* functionality for a fixed event during run-time, but the setup is in principle fit to perform all the aforementioned kinds of evolution. This is because we model superimposed code blocks and events as object structures.

**Language constructs for MCI** The syntax of a corresponding language extension is defined in Fig. 4. (One can for example extend the Java syntax accordingly.) The principle language construct is an expression form "**superimpose** *exp* **onto** *eve*", which allows the programmer to register additional functionality *exp* for future method calls that are characterised by *eve*. The syntactical domain *eve* is defined such that events are of the following form:

$$point : pat \rightarrow pat'$$

Here, *point* is either **dispatch**, or **enter**, or **exit**, while *pat* and $pat'$ denote patterns for sites such as "**class** *Foo* **&&** **method** *bar*". These are meant to constrain intercepted caller and callee sites on the basis of classes, method names, and others. The remaining expression forms **caller**, **callee**, **argument** and **result** are trivial, but still they are quite essential. These forms allow superimposed functionality to interact with the method call's context.

| Expression forms | | | |
|---|---|---|---|
| $exp$ | $=$ | $\cdots$ | |
| | $\mid$ | **superimpose** $exp$ **onto** $eve$ | (Perform MCI registration) |
| | $\mid$ | **caller** | (Access caller object) |
| | $\mid$ | **callee** | (Access callee object) |
| | $\mid$ | **argument** $vn$ | (Access method argument) |
| | $\mid$ | **result** | (Access tentative result) |
| **Events for superimposition** | | | |
| $eve$ | $=$ | $point$: $pat \rightarrow pat$ | (Point: caller → callee) |
| $point$ | $=$ | **dispatch** | (After VMT dispatch) |
| | $\mid$ | **enter** | (Before method body) |
| | $\mid$ | **exit** | (After method body) |
| **Patterns for sites** | | | |
| $pat$ | $=$ | * | (Every site) |
| | $\mid$ | **object** $exp$ | (Object as site) |
| | $\mid$ | **class** $cn$ | (Class of site) |
| | $\mid$ | **descendant** $cn$ | (... including subclasses) |
| | $\mid$ | **method** $mn$ | (Method name of site) |
| | $\mid$ | **result** $\tau$ | (Site by result type) |
| | $\mid$ | **argument** $\tau$ $vn$ | (Site by argument) |
| | $\mid$ | $pat$ && $pat$ | (Conjunction) |
| | $\mid$ | $pat$ \|\| $pat$ | (Disjunction) |
| | $\mid$ | ! $pat$ | (Negation) |

Figure 4: Syntax for method-call interception (MCI)

In Fig. 5, we show a simple example that illustrates the MCI constructs. In lines 2–9, we define a class `DoIt` with a method for the infamous `fac`torial function. The method body contains one method call, which happens to be a recursive call to `fac` itself. In lines 12–22, we define another class `WatchIt` which superimposes a counting action on calls of the `fac`torial method, and invokes `fac(10)`. This is explained in all detail as follows. In line 13, we declare the corresponding `counter`. In line 16, we construct a `DoIt` object and store the object reference in `myobj`. In lines 17–18, we superimpose an `inc`rement statement on `counter` onto method calls with `myobj` as callee. The fact that all possible caller objects are acceptable is expressed by the pattern "*". The callee object is specifically defined as "**object** `myobj`". Since we simply count method calls, the chosen point **dispatch** does not really matter. In lines 19–20, we first invoke the `fac`torial method, and then we print the `counter`'s value.

**Aspect-oriented programming** It is worth mentioning that MCI instantiates the notion of aspect-oriented programming (AOP; [31, 3, 18]). For completeness' sake, this link can be substantiated on the basis of a correspondence between the MCI and AOP terminology as follows:

```
1  // A class used as a target for MCI
2  class DoIt {
3   public int fac (int n) {
4    if (n > 0)
5     { return (n * fac(n-1)); }
6    else
7     { return 1; }
8   }
9  }
10
11 // A class hosting the superimposition
12 class WatchIt {
13  int counter = 0;
14
15  public void experiment () {
16   DoIt myobj = new DoIt;
17   superimpose counter = counter+1
18          onto dispatch: * -> object myobj;
19   fac(10);
20   System.out.println(counter+" calls done.");
21  }
22 }
```

Figure 5: Logging invocations of the factorial function

- In AOP for OO languages, it is a prominent kind of aspect to add functionality to methods quite similarly to MCI. For example, the terminology of AspectJ [46, 30, 32] offers the term 'advice' for such additional code.
- AOP involves a weaving process for the actual injection of advice into pre-existing code. Weaving can occur at compile-time, run-time, and others. MCI favours run-time weaving because it aims at run-time system adaptation.
- AOP is based on a join-point model, which defines the points along the execution of a program that can be possibly addressed by crosscutting concerns. MCI is restricted to method calls, but it distinguishes some fine points along a method call's execution.
- AOP also involves means of quantification for the join points to be affected by an aspect. The AOP term 'pointcut' refers to a collection of join points very much in the sense of MCI events.
- The presented MCI constructs are very simple, and they do not require any special abstraction form such as 'aspects' in AspectJ. It is not very difficult to extend MCI so that it covers further pointcut predicates or forms of advice, e.g., **around** advice or **cflow** join points as present in AspectJ.

**Objective of the paper** In the first part of the paper, we will present a concise static and dynamic semantics of MCI in the context of class-based, statically typed, compiled languages such as Java. In the second part of the paper, we follow a semantics-directed approach to language implementation. We will indeed use the semantics as a guideline for the implementation.[1] We argue that a semantics-directed implementation of MCI leads to the following combination of desirable properties:

- Simplicity of the implementational model.
- Run-time adaptation capabilities.
- Static type safety.
- Separate compilation.
- Reasonable performance.

By "a simple model" we mean that we enable MCI by simple source-code instrumentation as opposed to load-time weaving, run-time reflection, designated virtual machines, or just-in-time compilers. By "run-time adaptation capabilities" we mean that functionality can be superimposed onto a system that is already up and running. This has been identified as crucial expressiveness for rapid prototyping [40], dynamic component coordination [37], run-time system (re-) configuration [2], and others. By "static type safety" we mean the usual formula 'nothing can go wrong', i.e., added functionality is executed in a statically typed environment while there are type-safe means to interact with the objects involved in the actual method call, namely the caller, the callee, the arguments, and the tentative result. By "separate compilation" we mean that each class of a program can be compiled once and for all without any kind of closed-world assumption. Finally, by "reasonable performance" we mean that we get reasonably close to static weaving approaches. So the contribution of the paper is one of synthesis.

**Road-map of the paper** There are four technical sections:

- In Section 2, we present the essentials of the static and dynamic semantics of MCI. This is meant to enable a semantics-directed approach to the implementation of MCI.
- In Section 3, a naive implementational model for MCI is presented. The MCI language constructs are expanded to plain Java code. Static type safety is established prior to expansion. Superimposed code blocks and corresponding MCI events are maintained in an object serving as MCI registry. Method calls are instrumented to allow for interception, and to invoke the MCI registry to this end. The instrumented code can be processed by a Java compiler.

[1] An archive with our prototypical implementation of MCI together with a benchmarking environment is available for download on the paper's web-site: `http://www.cwi.nl/~ralf/sdmci/`. The actual implementation covers a subset of Java and it implements MCI via source-code instrumentation. The component for instrumentation is implemented in Prolog.

- In Section 4, several refinements of the naive implementational model are described. These are the principles that we will use: *caching* to memoise MCI inquiries, *bypassing* to shortcut MCI inquiries via switching logic, *class-based interception* as opposed to an initially object-based scheme, *laziness* to avoid unnecessary computations, and *wrapping* to reduce code-size increase for instrumentation.
- In Section 5, we discuss benchmarks. We cover a number of variation points for programs that use MCI (or maybe do not use it). This allows us to demonstrate the merits of the various refinements, and to compare our approach to other technology for system adaptation.

The remaining sections are compulsory.

- In Section 6, related work is reviewed.
- In Section 7, the paper is concluded.

**Acknowledgement**: The first author was supported in part by the German and Slovenian governments in the project SVN 99/028 ("*Adaptive, generic, and aspect-oriented language definitions*"), and by the Dutch research organisation NWO in the project 612.014.006 ("*Generation of program transformation systems*"). The first author gave presentations on the subject at the 55th IFIP WG 2.1 meeting ("*Algorithmic languages and calculi*") in Cochabamba, Bolivia, January 2001, at the Workshop Aspekt-Orientierung in Paderborn, Germany, May 2001, and at the 1st International Conference on Aspect-Oriented Software Development in Twente, The Netherlands, April 2002. The authors appreciated discussions with Günter Kniesel, Wolfgang Lohmann, and Günter Riedewald. The authors are very grateful for the very detailed and useful remarks by the three anonymous referees of the IEE Proceedings – Software; Special issue on "Unanticipated Software Evolution".

# 2 Semantics of MCI

We will now define the static and dynamic semantics of MCI. Our semantics definition is given in the style of Natural semantics [24, 16]. In the dynamic semantics part, we will present the dynamic semantics of the MCI constructs as well as the semantics of MCI-enabled method calls. (For brevity, we do not discuss the dynamic semantics of normal object-oriented constructs, which is prerequisite for MCI. The reader is referred to [34] for such semantical issues and to [41] for an executable specification of an OO language with MCI.) The dynamic semantics extends a normal object store to also hold an MCI registry for the superimposed functionality and the associated events. As an important detail of the dynamic semantics, we will need to specify a look-up operation for superimposed functionality. This includes a predicate to test if a pattern as it occurs as part of an MCI event 'intercepts' the caller or callee sites encountered at run-time. In the rest of the section, we will describe the static semantics of the MCI constructs. This includes provisions to guarantee the type-safe execution of superimposed functionality. To this end, we analyse MCI events such that we can make conservative assumptions about the context of an intercepted method call.

## 2.1 Dynamic semantics of MCI constructs

We assume an expression-oriented language as opposed to a separation of expressions and statements. That is, all kinds of constructs are expression forms, which might however involve side effects. In particular, method calls and MCI constructs are expression forms. We assume the following, reasonably standard judgement to define the meaning of expressions:

$$T, \Sigma, \theta, \eta \vdash \mathit{exp} \Rightarrow \upsilon, \Sigma'$$

The arguments of the judgement are explained as follows:

- $T$ comprises the normal method table, but also reflective information that will be needed for MCI, e.g., the subclassing relation or the class interfaces.
- $\Sigma$ and $\Sigma'$ model the store before and after evaluation. The store comprises the objects as field-to-value mappings, and the MCI registry as a list of event-functionality pairs. We also maintain the class of each object in the store.
- $\theta$ is the symbolic environment binding all kinds of special identifiers to their values. That is, $\theta$ comprises the object reference for **this**, but also the values for the MCI constructs that refer to caller, callee and others.
- $\eta$ is the variable environment binding method arguments and local variables to their values. (Note that we use a read-only environment. Hence, there are no assignments to local variables, but one would define such variables in "let ... in ..." expressions.)
- $\mathit{exp}$ is the expression subject to interpretation.
- $\upsilon$ is the result of expression evaluation.

In Fig. 6, the deduction rules for the MCI constructs are shown.[2] We will first explain the rule [mci]. The premises preprocess the MCI event, and then assemble an entry that is added to the MCI registry. A descriptor $\delta$, which points to the added entry, is the result of evaluation. (Subsequent operations can relate to the entry via the value $\delta$ as to enable the evolution scenarios from the introduction.) In more detail, the first two premises of [mci] refer to a helper judgement for the registration-time simplification of patterns. This judgement basically performs the following simplification:

$$\mathbf{object}\ \mathit{exp} \ \leadsto\ \overline{\mathbf{object}}\ \rho$$

Here, $\rho$ denotes the object reference as the result of evaluating $\mathit{exp}$. We use an extra constructor $\overline{\mathbf{object}}$. This simplification models that expressions inside patterns are evaluated at registration time as opposed to repeated evaluation at inquiry time. The

[2]Notation for deduction rules: We use $\pi_i(x)$ to denote projection for tuples. In $\pi_i(x)$, the $i$ is the index of some component, or the unambiguous type name of it. For example, in rule [mci], $\pi_\rho(\theta)$ and $\pi_{mn}(\theta)$ mean that we retrieve the object reference and the method name from the symbolic environment $\theta$ via projection. We use the notation $f \,@\, x = y$ to apply a mapping $f$ to a value $x$ which results in a value $y$. We use "$\perp$" to extend some domains by an undefined value. Normal meta-variables for the domains do not match with "$\perp$" by convention. "Don't care" variables, i.e., "_", match with all values including "$\perp$".

*Evaluation of expressions* $\boxed{T, \Sigma, \theta, \eta \vdash exp \Rightarrow v, \Sigma'}$

$$\frac{\begin{array}{ll} & T, \Sigma, \theta, \eta \vdash pat_1 : pat'_1, \Sigma' \\ \wedge & T, \Sigma', \theta, \eta \vdash pat_2 : pat'_2, \Sigma'' \\ \wedge & \kappa = \langle point, pat'_1, pat'_2 \rangle \\ \wedge & \alpha = \langle exp, \pi_\rho(\theta), \pi_{mn}(\theta), \eta \rangle \\ \wedge & \mathsf{add}(\Sigma'', \kappa, \alpha) \Rightarrow \delta, \Sigma''' \end{array}}{T, \Sigma, \theta, \eta \vdash \textbf{superimpose } exp \textbf{ on } point\text{: } pat_1 \rightarrow pat_2 \Rightarrow \delta, \Sigma'''} \quad [\mathsf{mci}]$$

$$T, \Sigma, \langle \rho, mn, \langle \rho', _, _, _ \rangle \rangle, \eta \vdash \textbf{caller} : \rho', \Sigma \quad [\mathsf{caller}]$$

$$T, \Sigma, \langle \rho, mn, \langle _, \rho', _, _ \rangle \rangle, \eta \vdash \textbf{callee} : \rho', \Sigma \quad [\mathsf{callee}]$$

$$\frac{\eta' \mathbin{@} vn = v}{T, \Sigma, \langle \rho, mn, \langle _, _, \eta', _ \rangle \rangle, \eta \vdash \textbf{argument } vn : v, \Sigma} \quad [\mathsf{argument}]$$

$$T, \Sigma, \langle \rho, mn, \langle _, _, _, v \rangle \rangle, \eta \vdash \textbf{result} : v, \Sigma \quad [\mathsf{result}]$$

Figure 6: Dynamic semantics of MCI constructs

next two premises in rule [mci] construct the run-time representations of the event (cf. $\kappa$) and the superimposed functionality (cf. $\alpha$). In the construction of $\kappa$, we simply combine the preprocessed ingredients of an MCI event in a triple. In the construction of $\alpha$, we annotate the superimposed expression *exp* with the object reference $\pi_\rho(\theta)$ (for **this**), the name $\pi_{mn}(\theta)$ of the currently executing method, and the variable environment $\eta$. This makes sure that *exp* can be executed later at interception time in its original registration-time environment. Also, by making available **this** from the registration context, the superimposed functionality can share state between subsequent executions. The last premise in rule [mci] adds the superimposition to the store, while it assigns a unique descriptor $\delta$ to it in case this superimposition needs to evolve.

The remaining deduction rules in Fig. 6 are trivial. They interpret the various constructs for interaction with a method call's context. In the rules, we simply look up the relevant piece from the symbolic environment. The way we use the symbolic environment, its structure is revealed as follows. At the top level, it is a triple $\langle \rho, mn, æ \rangle$. Here, $\rho$ is the object reference for **this**, while $mn$ is the name of the currently executing method, and *æ* is the environment assumed during the execution of superimposed functionality. This auxiliary environment is in turn a quadruple with one component for each construct **caller**, **callee**, **argument** and **result**. During the execution of normal code, the component *æ* takes the form "$\perp$" as we will see later in the rule for method calls. When superimposed functionality is executed, some of these components might be defined. At all points, at least the components for caller and callee are defined. At the points *enter* and *exit*, the arguments of an intercepted method call are defined, too. Only at the point *exit*, the component for the tentative result is defined.

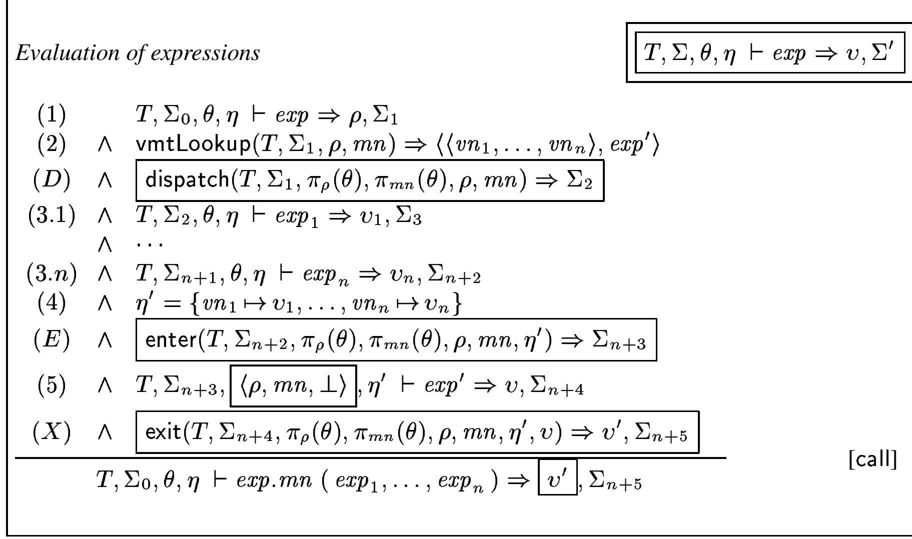


Figure 7: Dynamic semantics of MCI-enabled method calls

## 2.2 Dynamic semantics of MCI-enabled method calls

We are now in the position to define the meaning of method calls including the execution of superimposed functionality. In Fig. 7, we give the deduction rule for method calls while we box all premises and parameters that relate to MCI. There are three premises that deal with the points *dispatch*, *enter* and *exit*. The way the store is threaded (cf. $\Sigma_0$,..., $\Sigma_{n+5}$), one can clearly see how the execution of superimposed functionality (if any) is interleaved with the normal steps of a method call.

In premise (1), the callee $exp$ is evaluated resulting in an object reference $\rho$. In premise (2), the virtual method-table look-up is performed to retrieve the relevant method body and its formal parameters. (Class-based method look-up would need to query the method table, whereas object-based look-up would need to query the object store.) In premise (D), a helper judgement dispatch executes functionality that was superimposed onto this point. In the premises (3.1)–(3.n), the actual parameters are evaluated. In premise (4), the variable environment $\eta'$ for the method execution is constructed by binding actual to formal parameters. In premise (E), execution of functionality for the enter point commences. In premise (5), the body of the called method is executed. Notice the symbolic environment $\langle \rho, mn, \perp \rangle$ with $\rho$ is the called object (i.e., the object bound to **this**), with $mn$ as the name of the called method. The third component $\perp$ normally specifies the environment for the execution of superimposed functionality, which is however undefined here since we are about to execute a normal method body. Finally, in premise (X), execution of functionality for the exit point commences.

This semantics suggests separate compilation because we do not assume that the method table is adapted in any way to resemble superimposition. We rather model that superimpositions are placed in the normal store also used for mutable objects.

*The* **dispatch** *point* $\boxed{\mathsf{dispatch}(T, \Sigma, \rho, mn, \rho', mn') \Rightarrow \Sigma'}$

$$\frac{\begin{array}{ll} & \mathsf{mciLookup}(T, \Sigma_0, \mathbf{dispatch}, \rho, mn, \rho', mn') \Rightarrow \langle \alpha_1, \ldots, \alpha_n \rangle \\ \wedge & \alpha_1 = \langle exp_1, \rho_1, mn_1, \eta_1 \rangle \\ \wedge & \theta_1 = \langle \rho_1, mn_1, \langle \rho, \rho', \bot, \bot \rangle \rangle \\ \wedge & T, \Sigma_0, \theta_1, \eta_1 \vdash exp_1 \Rightarrow \upsilon_1, \Sigma_1 \\ \wedge & \cdots \\ \wedge & \alpha_n = \langle exp_n, \rho_n, mn_n, \eta_n \rangle \\ \wedge & \theta_n = \langle \rho_n, mn_n, \langle \rho, \rho', \bot, \bot \rangle \rangle \\ \wedge & T, \Sigma_{n-1}, \theta_n, \eta_n \vdash exp_n \Rightarrow \upsilon_n, \Sigma_n \end{array}}{\mathsf{dispatch}(T, \Sigma_0, \rho, mn, \rho', mn') \Rightarrow \Sigma_n} \quad [\mathsf{dispatch}]$$

*The* **enter** *point* $\boxed{\mathsf{enter}(T, \Sigma, \rho, mn, \rho', mn', \eta) \Rightarrow \Sigma'}$

$$\frac{\begin{array}{ll} & \mathsf{mciLookup}(T, \Sigma_0, \mathbf{enter}, \rho, mn, \rho', mn') \Rightarrow \langle \alpha_1, \ldots, \alpha_n \rangle \\ \wedge & \alpha_1 = \langle exp_1, \rho_1, mn_1, \eta_1 \rangle \\ \wedge & \theta_1 = \langle \rho_1, mn_1, \langle \rho, \rho', \eta, \bot \rangle \rangle \\ \wedge & T, \Sigma_0, \theta_1, \eta_1 \vdash exp_1 \Rightarrow \upsilon_1, \Sigma_1 \\ \wedge & \cdots \\ \wedge & \alpha_n = \langle exp_n, \rho_n, mn_n, \eta_n \rangle \\ \wedge & \theta_n = \langle \rho_n, mn_n, \langle \rho, \rho', \eta, \bot \rangle \rangle \\ \wedge & T, \Sigma_{n-1}, \theta_n, \eta_n \vdash exp_n \Rightarrow \upsilon_n, \Sigma_n \end{array}}{\mathsf{enter}(T, \Sigma_0, \rho, mn, \rho', mn', \eta) \Rightarrow \Sigma_n} \quad [\mathsf{enter}]$$

*The* **exit** *point* $\boxed{\mathsf{exit}(T, \Sigma, \rho, mn, \rho', mn', \eta, \upsilon) \Rightarrow \upsilon', \Sigma'}$

$$\frac{\begin{array}{ll} & \mathsf{mciLookup}(T, \Sigma_0, \mathbf{exit}, \rho, mn, \rho', mn') \Rightarrow \langle \alpha_1, \ldots, \alpha_n \rangle \\ \wedge & \alpha_1 = \langle exp_1, \rho_1, mn_1, \eta_1 \rangle \\ \wedge & \theta_1 = \langle \rho_1, mn_1, \langle \rho, \rho', \eta, \upsilon_0 \rangle \rangle \\ \wedge & T, \Sigma_0, \theta_1, \eta_1 \vdash exp_1 \Rightarrow \upsilon_1, \Sigma_1 \\ \wedge & \cdots \\ \wedge & \alpha_n = \langle exp_n, \rho_n, mn_n, \eta_n \rangle \\ \wedge & \theta_n = \langle \rho_n, mn_n, \langle \rho, \rho', \eta, \upsilon_{n-1} \rangle \rangle \\ \wedge & T, \Sigma_{n-1}, \theta_n, \eta_n \vdash exp_n \Rightarrow \upsilon_n, \Sigma_n \end{array}}{\mathsf{exit}(T, \Sigma_0, \rho, mn, \rho', mn', \eta, \upsilon_0) \Rightarrow \upsilon_n, \Sigma_n} \quad [\mathsf{exit}]$$

Figure 8: Dynamic semantics of superimposed functionality

### 2.3 Dynamic semantics of superimposed functionality

In the interpretation of MCI-enabled method calls, we used helper judgements $\mathsf{dispatch}$, $\mathsf{enter}$ and $\mathsf{exit}$ executing functionality that was registered for the points. These judgements are defined in Fig. 8. The three judgements only differ regarding details of environment construction and their involvement with the tentative result of the intercepted method call. The first premise always models look-up of functionality on the basis of the present caller site $\rho$, $mn$ and callee site $\rho'$, $mn'$. (The basic judgement $\mathsf{mciLookup}$ is discussed below.) Several pieces $\alpha_1, \ldots, \alpha_n$ might need to be executed for a given point. For each such $\alpha_i$, a symbolic environment $\theta_i$ of the form $\langle \rho_i, mn_i, æ_i \rangle$ is constructed to be used for the execution of the expression part $exp_i$ of $\alpha_i$. The components $\rho_i$ and $mn_i$ carry over from $\alpha_i$. The auxiliary environment $æ_i$ depends on the point at hand. At the point *dispatch*, $æ$ takes the form $\langle \rho, \rho', \bot, \bot \rangle$ to reflect that only caller and callee are defined. Both the arguments of the call and its tentative result are simply not yet known at this point. At the points *enter* and *exit*, the arguments of the call are also defined, but the result is defined at the point *exit* only. Rule [exit] clarifies how the tentative result $v_0$ can be revised through $v_1, \ldots, v_n$ by the execution of the pieces $exp_1, \ldots, exp_n$ of superimposed functionality.

The basic judgement for looking up superimposed functionality is typed as follows:

$$\mathsf{mciLookup}(T, \Sigma, point, \rho, mn, \rho', mn') \Rightarrow \alpha^*$$

Here $\rho, mn$ refer to the caller site, while $\rho', mn'$ refer to the callee site. In Fig. 9, we specify the judgement $\mathsf{mciLookup}$ and a helper $\mathsf{intercept}$. In the definition of $\mathsf{mciLookup}$, we express that there can be zero or more matching event-functionality pairs for the given join point. We compute this sequence $\alpha^*$ as a list comprehension by filtering the registry part of the store. To see if an MCI event $\kappa$ fits we have to check if its patterns intercept the given sites for caller and callee. This is modelled by the helper judgement $\mathsf{intercept}$. There is a deduction rule for each form of a pattern, which describes the constraints on the sites to be intercepted. For example:

- "$*$" intercepts $\langle \rho, mn \rangle$ for all $\rho$ and $mn$.
- "$\overline{\textbf{object}}\ \rho'$" does not intercept $\langle \rho, mn \rangle$ if $\rho \neq \rho'$.
- "**class** $cn$" intercepts $\langle \rho, mn \rangle$ if $\rho$ is of class $cn$.

The judgement $\mathsf{intercept}(T, \Sigma, \rho, mn, pat)$ involves parameters for the method table and the store, which are needed for some forms of patterns. The deduction rules manifest that some amount of type information must be available at run-time. Firstly, given an object reference $\rho$, the premise $\mathsf{class}(\Sigma, \rho) \Rightarrow cn$ models retrieval of the class $cn$ of $\rho$ from the store $\Sigma$ (cf. rule [class] etc.). Secondly, given two classes $cn$ and $cn'$, the premise $\mathsf{descendant}(T, cn, cn')$ models a test if $cn$ is a descendant class of $cn'$ (in terms of the reflexive, transitive closure of the subclassing relation). We can reason about result types and formal parameters of methods in a similar manner (cf. rules [result] and [argument]). So we end up with the following, very simple reflection API:

*Look up superimposed functionality* $\boxed{\mathsf{mciLookup}(T, \Sigma, point, \rho, mn, \rho', mn') \Rightarrow \alpha^*}$

$$\frac{\alpha^* = \left[ \alpha \;\middle|\; \begin{array}{ll} & \langle \kappa, \alpha \rangle \in \Sigma \\ \wedge & \kappa = \langle point, pat, pat' \rangle \\ \wedge & \mathsf{intercept}(T, \Sigma, \rho, mn, pat) \\ \wedge & \mathsf{intercept}(T, \Sigma, \rho', mn', pat') \end{array} \right]}{\mathsf{mciLookup}(T, \Sigma, point, \rho, mn, \rho', mn') \Rightarrow \alpha^*} \quad [\mathsf{mciLookup}]$$

*Test if pattern intercepts site* $\boxed{\mathsf{intercept}(T, \Sigma, \rho, mn, pat)}$

$$\mathsf{intercept}(T, \Sigma, \rho, mn, *) \quad [\mathsf{all}]$$

$$\mathsf{intercept}(T, \Sigma, \rho, mn, \overline{\mathbf{object}}\ \rho) \quad [\mathsf{object}]$$

$$\frac{\mathsf{class}(\Sigma, \rho) \Rightarrow cn}{\mathsf{intercept}(T, \Sigma, \rho, mn, \mathbf{class}\ cn)} \quad [\mathsf{class}]$$

$$\frac{\begin{array}{ll} & \mathsf{class}(\Sigma, \rho) \Rightarrow cn' \\ \wedge & \mathsf{descendant}(T, cn', cn) \end{array}}{\mathsf{intercept}(T, \Sigma, \rho, mn, \mathbf{descendant}\ cn)} \quad [\mathsf{descendant}]$$

$$\mathsf{intercept}(T, \Sigma, \rho, mn, \mathbf{method}\ mn) \quad [\mathsf{method}]$$

$$\frac{\begin{array}{ll} & \mathsf{class}(\Sigma, \rho) \Rightarrow cn \\ \wedge & \mathsf{argument}(T, cn, mn, vn) \Rightarrow \tau \end{array}}{\mathsf{intercept}(T, \Sigma, \rho, mn, \mathbf{argument}\ \tau\ vn)} \quad [\mathsf{argument}]$$

$$\frac{\begin{array}{ll} & \mathsf{class}(\Sigma, \rho) \Rightarrow cn \\ \wedge & \mathsf{result}(T, cn, mn) \Rightarrow \tau \end{array}}{\mathsf{intercept}(T, \Sigma, \rho, mn, \mathbf{result}\ \tau)} \quad [\mathsf{result}]$$

$$\frac{\begin{array}{ll} & \mathsf{intercept}(T, \Sigma, \rho, mn, pat_1) \\ \wedge & \mathsf{intercept}(T, \Sigma, \rho, mn, pat_2) \end{array}}{\mathsf{intercept}(T, \Sigma, \rho, mn, pat_1\ \mathbf{\&\&}\ pat_2)} \quad [\mathsf{and}]$$

$$\frac{\exists i \in \{1, 2\}.\ \mathsf{intercept}(T, \Sigma, \rho, mn, pat_i)}{\mathsf{intercept}(T, \Sigma, \rho, mn, pat_1\ ||\ pat_2)} \quad [\mathsf{or}]$$

$$\frac{\neg\ \mathsf{intercept}(T, \Sigma, \rho, mn, pat)}{\mathsf{intercept}(T, \Sigma, \rho, mn, \mathbf{!}\ pat)} \quad [\mathsf{not}]$$

Figure 9: The test for interception

- $\mathsf{class}(\Sigma, \rho) \Rightarrow cn$
- $\mathsf{descendant}(T, cn, cn')$
- $\mathsf{result}(T, cn, mn) \Rightarrow \tau$
- $\mathsf{argument}(T, cn, mn, vn) \Rightarrow \tau$

The first member is available anyway in an object-oriented language model with type-safe cast or instanceOf test. The other members reify the subclass relationship and the class interfaces. This is very basic read-only reflection.

## 2.4 Static semantics of MCI constructs

We need to impose typing rules on the MCI constructs so that no unsafe assumptions are made by superimposed functionality when accessing caller, callee, arguments of the call, or its tentative result. For convenience, we define the typing judgement in a way that it follows the same scheme as the one for interpretation. In $T, \Sigma, \theta, \eta \vdash exp : \tau$, the parameters deal now with types as opposed to values in the dynamic semantics. That is, $\tau$ is the type of the expression $exp$, $T$ is a table with the method signatures, $\Sigma$ maps fields to types, $\theta$ is the symbolic environment with types of **this**, **caller**, etc., and $\eta$ is the variable environment with the types of method arguments and local variables.

Here is a well-typed and an ill-typed superimposition:

- Consider the following superimposition:

  $$\begin{array}{l}\textbf{superimpose}\ \textbf{callee}.\mathit{hook}() \\ \ \textbf{onto dispatch}: * \rightarrow (\textbf{class}\ \mathit{Foo}\ \textbf{\&\&}\ \textbf{method}\ \mathit{bar})\end{array}$$

  That is, each call to a method *bar* of objects of class *Foo* will be intercepted, and the callee's method *hook* is invoked. This superimposition is well-typed if the interface of *Foo* provides the methods *bar* and *hook* of suitable types.

- Consider a slightly different superimposition:

  $$\begin{array}{l}\textbf{superimpose}\ \boxed{\textbf{caller}}.\mathit{hook}() \\ \ \textbf{onto dispatch}: * \rightarrow (\textbf{class}\ \mathit{Foo}\ \textbf{\&\&}\ \textbf{method}\ \mathit{bar})\end{array}$$

  According to our type system, this superimposition is ill-typed because the caller is insufficiently constrained by "$*$". It is not established that arbitrary callers implement methods *bar* and *hook* (unless the base class would offer them).

In Fig. 10, the typing rules for all MCI constructs are shown. The rules for **caller**, **callee**, **argument** and **result** are again trivial because we only need to select the appropriate parts of the symbolic environment. The rules [dispatch], [enter] and [exit] deal with superimposition for the three different MCI points. We assume a number of helper judgements safe..., which are meant to determine conservative assumptions about caller, callee, arguments of the method call, and tentative result. The rules also

*Well-typedness of expressions* $\boxed{T, \Sigma, \theta, \eta \vdash exp : \tau}$

$$\frac{\begin{array}{ll} & \mathsf{safeClass}(T, \Sigma, \theta, \eta, pat) \Rightarrow cn \\ \wedge & \mathsf{safeClass}(T, \Sigma, \theta, \eta, pat') \Rightarrow cn' \\ \wedge & \theta' = \langle \pi_{cn}(\theta), \pi_{mn}(\theta), \langle cn, cn', \bot, \bot \rangle \rangle \\ \wedge & T, \Sigma, \theta', \eta \vdash exp : \tau \end{array}}{T, \Sigma, \theta, \eta \vdash \textbf{superimpose } exp \textbf{ on dispatch}: pat \to pat' : \theta'} \quad [\mathsf{dispatch}]$$

$$\frac{\begin{array}{ll} & \mathsf{safeClass}(T, \Sigma, \theta, \eta, pat) \Rightarrow cn \\ \wedge & \mathsf{safeClass}(T, \Sigma, \theta, \eta, pat') \Rightarrow cn' \\ \wedge & \mathsf{safeArguments}(T, \Sigma, \theta, \eta, pat') \Rightarrow \eta' \\ \wedge & \theta' = \langle \pi_{cn}(\theta), \pi_{mn}(\theta), \langle cn, cn', \eta', \bot \rangle \rangle \\ \wedge & T, \Sigma, \theta', \eta \vdash exp : \tau \end{array}}{T, \Sigma, \theta, \eta \vdash \textbf{superimpose } exp \textbf{ on enter}: pat \to pat' : \theta'} \quad [\mathsf{enter}]$$

$$\frac{\begin{array}{ll} & \mathsf{safeClass}(T, \Sigma, \theta, \eta, pat) \Rightarrow cn \\ \wedge & \mathsf{safeClass}(T, \Sigma, \theta, \eta, pat') \Rightarrow cn' \\ \wedge & \mathsf{safeArguments}(T, \Sigma, \theta, \eta, pat') \Rightarrow \eta' \\ \wedge & \mathsf{safeResult}(T, \Sigma, \theta, \eta, pat') \Rightarrow \tau \\ \wedge & \theta' = \langle \pi_{cn}(\theta), \pi_{mn}(\theta), \langle cn, cn', \eta', \tau \rangle \rangle \\ \wedge & T, \Sigma, \theta', \eta \vdash exp : \tau \end{array}}{T, \Sigma, \theta, \eta \vdash \textbf{superimpose } exp \textbf{ on exit}: pat \to pat' : \theta'} \quad [\mathsf{exit}]$$

$$T, \Sigma, \langle cn, mn, \langle cn', _, _, _ \rangle \rangle, \eta \vdash \textbf{caller} : cn' \quad [\mathsf{caller}]$$

$$T, \Sigma, \langle cn, mn, \langle _, cn', _, _ \rangle \rangle, \eta \vdash \textbf{callee} : cn' \quad [\mathsf{callee}]$$

$$\frac{\eta' \,@\, vn = \tau}{T, \Sigma, \langle cn, mn, \langle _, _, \eta', _ \rangle \rangle, \eta \vdash \textbf{argument } vn : \tau} \quad [\mathsf{argument}]$$

$$T, \Sigma, \langle cn, mn, \langle _, _, _, \tau \rangle \rangle, \eta \vdash \textbf{result} : \tau \quad [\mathsf{result}]$$

Figure 10: Static semantics of MCI constructs

state that the type of a superimposition is the symbolic environment that was also used to check the superimposed functionality. This enables checking type safety of operations such as inclusion of extra intercepted calls or revision of the superimposed functionality. (Recall the evolution scenarios from the introduction.) The three deduction rules [dispatch], [enter], [exit] only differ in the provided environment components. At the point *dispatch*, the components for argument types of the method call and its result type must be left undefined. Argument types are defined from the point *enter* on. The type of the tentative result is defined at the point *exit* only.

*Determine class*

$$\boxed{\mathsf{safeClass}(T, \Sigma, \theta, \eta, pat) \Rightarrow cn}$$

$$\mathsf{safeClass}(T, \Sigma, \theta, \eta, *) \Rightarrow \top \qquad \text{[all]}$$

$$\frac{T, \Sigma, \theta, \eta \vdash exp : cn}{\mathsf{safeClass}(T, \Sigma, \theta, \eta, \mathbf{object}\ exp) \Rightarrow cn} \qquad \text{[object]}$$

$$\frac{\mathsf{classExists}(T, cn)}{\mathsf{safeClass}(T, \Sigma, \theta, \eta, \mathbf{class}\ cn) \Rightarrow cn} \qquad \text{[class]}$$

$$\frac{\mathsf{classExists}(T, cn)}{\mathsf{safeClass}(T, \Sigma, \theta, \eta, \mathbf{descendant}\ cn) \Rightarrow cn} \qquad \text{[descendant]}$$

$$\frac{\begin{array}{ll} & \mathsf{classExists}(T, cn) \\ \wedge & \mathsf{methodExists}(T, cn, mn) \end{array}}{\mathsf{safeClass}(T, \Sigma, \theta, \eta, \mathbf{method}\ mn) \Rightarrow \top} \qquad \text{[method]}$$

$$\frac{\begin{array}{ll} & \mathsf{classExists}(T, cn) \\ \wedge & \mathsf{methodExists}(T, cn, mn) \\ \wedge & \mathsf{argument}(T, cn, mn, vn) \Rightarrow \tau \end{array}}{\mathsf{safeClass}(T, \Sigma, \theta, \eta, \mathbf{argument}\ \tau\ vn) \Rightarrow \top} \qquad \text{[argument]}$$

$$\frac{\begin{array}{ll} & \mathsf{classExists}(T, cn) \\ \wedge & \mathsf{methodExists}(T, cn, mn) \\ \wedge & \mathsf{result}(T, cn, mn) \Rightarrow \tau \end{array}}{\mathsf{safeClass}(T, \Sigma, \theta, \eta, \mathbf{result}\ \tau) \Rightarrow \top} \qquad \text{[result]}$$

$$\frac{\begin{array}{ll} & \mathsf{safeClass}(T, \Sigma, \theta, \eta, pat_1) \Rightarrow cn_1 \\ \wedge & \mathsf{safeClass}(T, \Sigma, \theta, \eta, pat_2) \Rightarrow cn_2 \\ \wedge & cn = \begin{cases} cn_1, & \text{if } \mathsf{descendant}(T, cn_1, cn_2) \\ cn_2, & \text{if } \mathsf{descendant}(T, cn_2, cn_1) \\ \top, & \text{otherwise} \end{cases} \end{array}}{\mathsf{safeClass}(T, \Sigma, \theta, \eta, pat_1\ \&\&\ pat_2) \Rightarrow cn} \qquad \text{[and]}$$

$$\frac{\begin{array}{ll} & \mathsf{safeClass}(T, \Sigma, \theta, \eta, pat_1) \Rightarrow cn_1 \\ \wedge & \mathsf{safeClass}(T, \Sigma, \theta, \eta, pat_2) \Rightarrow cn_2 \\ \wedge & cn = \begin{cases} cn_1, & \text{if } \mathsf{descendant}(T, cn_2, cn_1) \\ cn_2, & \text{if } \mathsf{descendant}(T, cn_1, cn_2) \\ \top, & \text{otherwise} \end{cases} \end{array}}{\mathsf{safeClass}(T, \Sigma, \theta, \eta, pat_1 \mid\mid pat_2) \Rightarrow cn} \qquad \text{[or]}$$

$$\mathsf{safeClass}(T, \Sigma, \theta, \eta, \mathbf{!}\ _) \Rightarrow \top \qquad \text{[not]}$$

Figure 11: Conservative assumptions about caller and callee classes

It remains to define conservative assumptions about caller, callee, etc. that can be relied upon within superimposed functionality. Clearly, these assumptions have to be extracted from the MCI events. Conservatism is meant to avoid unsafe execution of superimposed functionality. To this end, we observe that patterns basically form conjunctions and disjunctions over more basic forms such as constraints on class names, subclassing relationships, method names, result types, formal parameter names and types. So the overall idea for extracting safe assumptions from such composites is to determine what basic constraints are promoted by the composite events. This is worked out for caller / callee classes in Fig. 11. The definitions for safe result types and safe formal parameters are omitted because they are very similar.

The judgement safeClass simply provides one deduction rule per form of pattern. The form "$*$" does not allow any specific assumption about a class that matches with this pattern. Hence, we assume "$\top$" (i.e., the class `Object` in Java), which is safe because the caller or callee object will at least meet this base-class interface. The form "**object** *exp*" allows for a better assumption for the class. That is, we can determine the type of the expression *exp*, which is an upper bound for the run-time type of *exp*. The form "**class** *cn*" immediately suggests a good assumption for the caller / callee class. As a kind of simple sanity check, the deduction rule comprises a premise to check that *cn* actually denotes a known class name (cf. classExists). Similarly, the form "**descendant** *cn*" also suggests the assumption *cn*, event though at run-time, a caller / callee object $\rho$ of a subclass could be encountered. (This is well in line with the normal subsumption principle.) For the forms "**method** *mn*", "**result** $\tau$" and "**argument** $\tau$ *vn*" we again resort to the conservative "$\top$" assumption, but we check that the given names and types are valid. It remains to define conjunction and disjunction. In a conjunction, we always prefer the least class if there is one. So dually, in a disjunction, we always prefer the greatest class if there is one.[3]

This completes the definition of the static and dynamic semantics of MCI.

# 3 Naive implementation of MCI

We employ source-code instrumentation for the implementation of MCI. This technique is generally suited for experiments in language design and implementation. However, it is not immediately clear whether instrumentation at compile time is sufficient or efficient in the context of run-time adaptation as favoured by MCI. The results of our implementational development indicate that instrumentation at compile time is indeed a valid option. However, refinements of the naive model from this section are necessary as discussed in the next section. We assume here that MCI is added to a class-based, statically typed, compiled language. Source-code examples are given in Java.

[3]Note that these definitions do not precisely agree with the algebraic notions of least upper bound and greatest lower bound. This is because we have a greatest element "$\top$" but no natural least element. So in both cases, it makes sense to resort to the weakest assumption "$\top$" if no specific class can be established.

Our development follows a semantics-directed approach [38, 35], but we do not assume the strongest form of this principle where a compiler would be just a different 'interpretation' of the dynamic semantics. Our goal is merely to preserve the link between semantics and language implementation in so far that the essential semantical concepts are traceable in the implementational model. This concerns these concepts: events, superimposed functionality, MCI registry, context of a method call, look-up of superimposed functionality, environment construction, and reification of class interfaces.

We opt for the following formula to define an implementational model for MCI:

- Static type safety is established at the level of the extended language — before source-code instrumentation compiles away the MCI constructs and turns them into plain OOP constructs. The static semantics from the previous section can be effectively implemented as is.
- We store event-functionality pairs in a designated object, say the *MCI registry*. The execution of the superimposition construct indeed extends the MCI registry. We instrument method calls to invoke the MCI registry, which in turn follows the semantics of interleaving an ordinary method call with actions for look-up, environment construction, and execution of superimposed functionality.
- Each superimposed code block is captured in a dedicated class that also supports a method for running the code. Thereby the code block can be passed to the registry as an object. MCI events are represented as object structures that support a method for the interception test.
- When invoking the MCI registry, we pass on the caller and callee objects, and the method arguments. Furthermore, reflective information for the corresponding class names, method names, and method signatures is passed on. This information is made available by source-code instrumentation.

We will detail this formula in the sequel where we skip the first item on static type safety because here the semantics and the implementation truly coincide.

## 3.1 The MCI registry

The run-time model for MCI employs a MCI registry; see Fig. 12 for an illustration. The registry can be invoked in the ways indicated by the incoming arrows. The code originating from a **superimpose** expression sends an *add* message along with arguments for the MCI event and the superimposed code block. There are designated classes *MCI_event* and *MCI_advice*. The code for MCI-enabled method calls uses methods *dispatch*, *enter* and *exit* to invoke the MCI registry at these points. In turn, the registry filters its entries to determine relevant superimposed code blocks, which are then ultimately executed. To this end, the *run* method of the class *MCI_advice* is invoked. For a reasonable run-time performance, several optimisations are mandatory as discussed in Sec. 4.

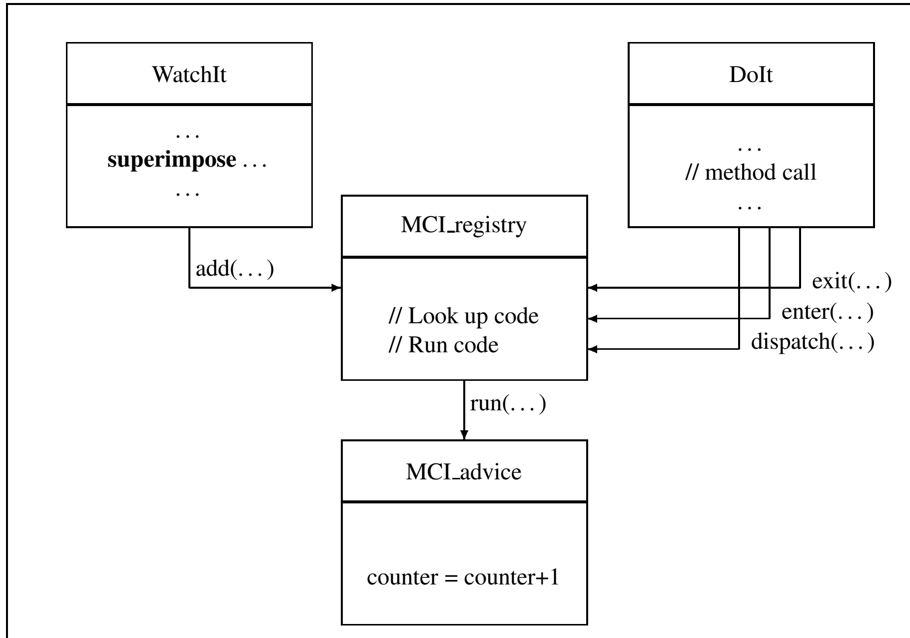
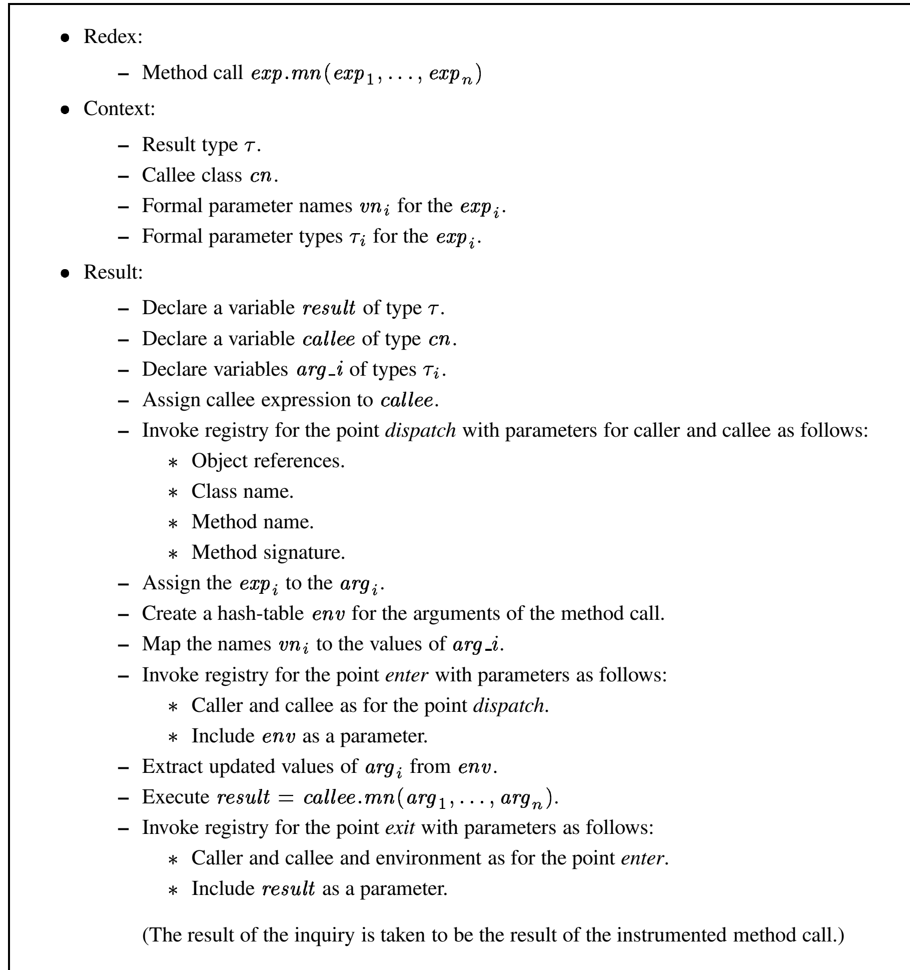



Figure 12: MCI run-time model based on an MCI registry (instantiated for Fig. 5)

## 3.2 Instrumentation of method calls

Each method call has to be expanded into a code sequence that interleaves the actual call with the inquiries that are sent to the MCI registry. The complete sequence also contains several steps related to the construction of the environment for superimposed functionality (if any). Extra provisions are needed if we want to allow the programmer to change the arguments of the call. In Fig. 13, we present the resulting scheme of source-code instrumentation. There are some ways to instrument method calls. The given scheme wraps calls rather than methods. (We will discuss method wrappers later.) The scheme is defined as a transformation rule with a redex, a context, and a result. By the redex we mean that the scheme is applied to a method call. By the context we mean that we are able to retrieve type-like information about the elements of the method call. In the result section of the rule, we define the sequence of declarations and statements that replace the original method call.

In Fig. 14, we show the instrumented `fac`torial method for the introductory example from Fig. 5. The actual method call is boxed and all inquiries that are sent to the MCI registry are printed in bold face. In lines 4–7, we declare variables that are specifically needed to enable MCI of the single method call in the `fac`torial function. In line 8, we compute the callee which trivially resolves to **this** here because the `fac`torial function is recursive. In lines 9–11, we invoke the registry to handle the `dispatch` point. The arguments provide all the needed details about caller and callee, namely the object reference, the name of the method, the result type, and the method's signature



- Redex:
  - Method call $exp.mn(exp_1, \ldots, exp_n)$
- Context:
  - Result type $\tau$.
  - Callee class $cn$.
  - Formal parameter names $vn_i$ for the $exp_i$.
  - Formal parameter types $\tau_i$ for the $exp_i$.
- Result:
  - Declare a variable $result$ of type $\tau$.
  - Declare a variable $callee$ of type $cn$.
  - Declare variables $arg_i$ of types $\tau_i$.
  - Assign callee expression to $callee$.
  - Invoke registry for the point *dispatch* with parameters for caller and callee as follows:
    - Object references.
    - Class name.
    - Method name.
    - Method signature.
  - Assign the $exp_i$ to the $arg_i$.
  - Create a hash-table $env$ for the arguments of the method call.
  - Map the names $vn_i$ to the values of $arg_i$.
  - Invoke registry for the point *enter* with parameters as follows:
    - Caller and callee as for the point *dispatch*.
    - Include $env$ as a parameter.
  - Extract updated values of $arg_i$ from $env$.
  - Execute $result = callee.mn(arg_1, \ldots, arg_n)$.
  - Invoke registry for the point *exit* with parameters as follows:
    - Caller and callee and environment as for the point *enter*.
    - Include $result$ as a parameter.

  (The result of the inquiry is taken to be the result of the instrumented method call.)

Figure 13: Scheme for source-code instrumentation of MCI-enabled method calls

`_fac_sig`. Here, we assume that classes are instrumented to hold signatures for all their methods. The next few lines 12–14 represent the evaluation of the method arguments and the construction of the argument environment. In lines 15–18, we invoke the registry to handle the `enter` point. Since the method arguments are meanwhile known and stored in `_env`, we also include `_env` in the inquiry message. In line 19, we ensure that we continue with the most up-to-date method argument for the actual computation of the `fac`torial because superimposed functionality might have changed the argument. In line 20, we perform the actual method call. In lines 21–25, we invoke the registry to handle the `exit` point. Here, we also pass on the tentative result from the original method call in line 20. The remaining lines 26–29 complete the definition of the `fac`torial function.

```
1  public int fac (int n)
2  {
3   if (n > 0) {
4     int _result = 0;         // tentative result
5     DoIt _callee = null;     // callee
6     Hashtable _env = null;  // argument environment
7     int _arg_1 = 0;          // side-affected argument
8     _callee = this;
9     MCI_registry.dispatch(
10      this,"fac","integer",this._fac_sig,
11      _callee,"fac","integer",_callee._fac_sig);
12    _arg_1 = n-1;
13    _env = new Hashtable(1,1);
14    _env.put("n",new Integer(_arg_1));
15     MCI_registry.enter(
16      this,"fac","integer",this._fac_sig,
17      _callee,"fac","integer",_callee._fac_sig,
18      _env);
19    _arg_1 = (_env.get("n")).intValue();
20    _result = _callee.fac(_arg_1);
21    _result = ((MCI_registry.exit(
22      this,"fac","integer",this._fac_sig,
23      _callee,"fac","integer",_callee._fac_sig,
24      _env,
25      _result))).intValue();
26    return (n * _result);
27  } else
28  { return 1; };
29 }
```

Figure 14: Java method readily instrumented for MCI

In Fig. 15, we illustrate how we provide the method signatures for the MCI implementation as required above. We basically reify method signatures.[4] That is, we add members to all classes, so that one can retrieve their method signatures. In the figure, this is illustrated for the class `DoIt` which only holds one method — the `fac`torial method. In line 3, we declare a static hash table to hold the signature for the method `fac`. Such a hash table would be needed for every method. In lines 5–8, we compute the hash table by storing the type information about the single argument of the `fac`torial method. This is done once and for all in the class. So there is no per-object or per-method-call overhead.

[4]In our implementational development, we do not make use of Java's reflection API because some information is not accessible anyway. We place all required reflective information explicitly in the instrumented code. This is also highly efficient because the information is made readily available as opposed to relying on the invocation of a reflection API.

```
class DoIt {

 public static Hashtable _fac_sig;

 static {
    _fac_sig = new Hashtable(1,1);
    _fac_sig.put("n","int");
 }

 public int fac (int n) {
   // see Fig. 14
 }

}
```

Figure 15: Java class with signature information

```
class WatchIt  {
 int counter = 0;

 public void experiment () {
  DoIt myobj = new DoIt;
  MCI_aspect _aspect1 = MCI_registry.add(
    new WatchIt_advice_1(this),
    new MCI_dispatch( new MCI_pat_all(),
                      new MCI_pat_object(myobj)) );
  fac(10);
  System.out.println(counter+" calls done.");
 }
}
```

Figure 16: Java representation of superimposition

## 3.3 Compilation of superimpositions

The **superimpose** form of expression is translated into an `add` message to be sent to the MCI registry. This is shown for our running example in Fig. 16; see lines 6–9. The MCI event is constructed as an object structure following closely the syntactical structure of events. (To represent events or 'pointcuts' as compound objects is standard practice in aspect-oriented frameworks; see, e.g., [40].) The superimposed code block is hosted by an auxiliary class `WatchIt_advice_1`, and so an instance of this class is constructed for inclusion in the message. The return type of the `add` message is `MCI_aspect`. This class implements some operations for the adaptation of the MCI registry — as needed for scenarios of evolution.

```
1  class WatchIt_advice_1 extends MCI_advice {
2
3    // Host of the advice
4    private WatchIt _this;
5
6    // Constructor to be invoked by superimposition
7    public WatchIt_advice_1 (WatchIt host) {
8      super(); _this = host;
9    }
10
11   // Run method to be invoked by registry
12   public Object run (Object caller,
13                      Object callee,
14                      Hashtable env,
15                      Object result) {
16     _this.counter = _this.counter+1;
17     return null;
18   }
19 }
```

Figure 17: Java representation of superimposed code block

The Java class `WatchIt_advice_1` with the superimposed code block from the `fac`torial example is shown in Fig. 17. The code for counting is boxed. (Instead of placing superimposed functionality in extra classes, we could also attempt to place methods with superimposed functionality in the class hosting the superimposition. This, however, will make it harder for the registry to run such code in a uniform way. This is because there might be several superimposed code blocks in one class. Alternatively, Java's inner classes could be used here to make the reference to the host of the superimposition explicit.) The class `WatchIt_advice_1` follows a very simple scheme. In line 4, we declare an instance variable to keep track of the host that issues the registration. It is a kind of virtual self reference. In lines 7–9, we declare the constructor for the code block. This constructor must be invoked with the **this** reference from the superimposition context. In lines 12–18, we declare the `run` method that executes the code block on request. The `run` method is prepared to receive all context information that might be relevant for the points *dispatch*, *enter* and *exit*. Hence, there are arguments for the caller, the callee, the arguments of the call, and the tentative result. (We could also use three variations on `run` instead.) Note that the Java representation is weekly typed in that these context arguments are simply of type `Object`, which is necessary for uniformity. Hence, the instrumented code will normally perform cast operations to access caller, callee, and others. This situation clarifies the contribution of the static semantics of MCI, which is more precise with regard to the typing of the context of an intercepted method call.

This completes the definition of the naive implementational model for MCI.

# 4 Refinements of the MCI implementation

The efficiency of the source-code instrumentation approach largely depends on the precise code template for method-call instrumentation, and on the implementation of the MCI registry. So we need to identify the idioms that help with achieving an efficient implementation. None of the ideas that we will discuss here are very original; several elements of our development also occur in the rich literature on Metaobject protocols and reflection [29, 12, 33, 15, 20, 45]. So the contribution of this section is more like a survey on optimisation techniques. While we will focus on the optimisation of the run-time efficiency, there is also a short discussion of the code-size issue. This is how we structure our survey:

**Caching** If the same kind of inquiry is performed several times, caching or memoisation pays off. MCI look-up is obviously an expensive operation — depending on the number of entries in the MCI registry and the structure of MCI events.

**Bypassing** There is a considerable overhead per method call because of the protocol for the invocation of the MCI registry. We can radically reduce this overhead by a switching logic to bypass MCI inquiries if possible.

**Class-based interception** The pattern form "**object** *exp*" basically suggests interception with an object-based precision. This challenges caching and bypassing considerably. We can use radically more efficient class-based inquiries on the basis of a semantics-preserving program transformation.

**Laziness** We can handle actions that form part of an inquiry in a lazy manner. In particular, the construction of the environment for the superimposed functionality can be performed on demand depending on the needs of the superimposition at hand.

**Wrapping** Rather than instrumenting method calls directly, one can also wrap methods. This implies that the code for instrumentation is only added once per method and not for each method call.

We will explain these techniques now in detail. Benchmarks will follow in Sec. 5.

## 4.1 Caching MCI inquiries

**Global cache** In the most basic setting, we use a global cache maintained by the MCI registry to speed up look-up. This run-time model is illustrated in Fig. 18. Most notably, we assume a *cache* object, and all inquiries that are sent to the MCI registry have to point out a key for caching. Then, the MCI registry *looks up* superimposed code blocks from the cache using the given key. Whenever the MCI registry is changed, it sends a *reset* message to the cache. This is necessary to enforce a consistent cache in the view of run-time adaptation capabilities. The reset operation does not necessarily need to reset the complete cache, but it can act more selectively depending on the impact of the system adaptation at hand.

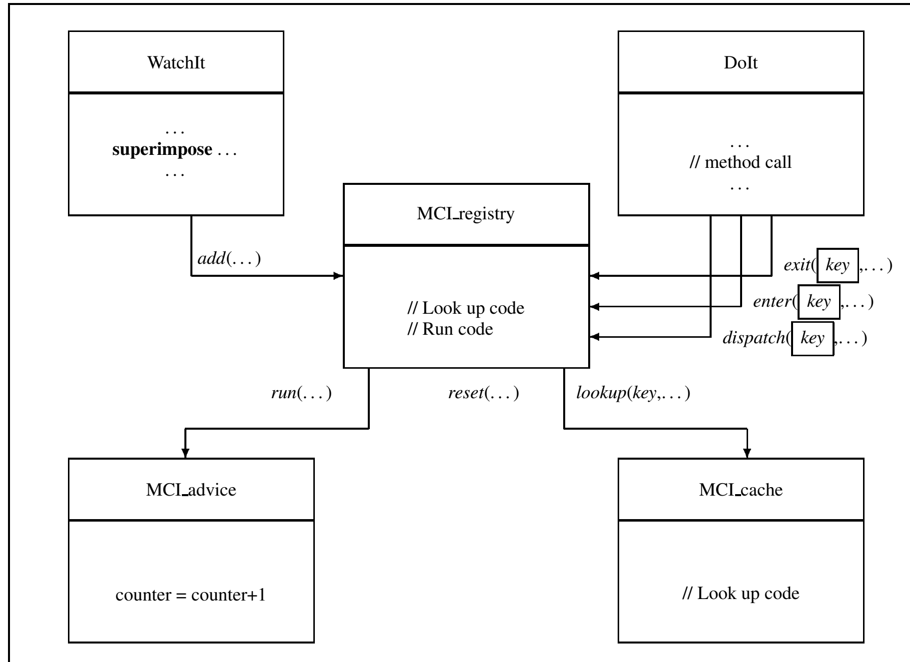



Figure 18: MCI run-time model with a global cache for the MCI registry

**Naive keys** Conceptually, the key needs to consist of the following components if we want to trust the result of cache look-up without further postprocessing:

- An MCI point (i.e., *dispatch* vs. *enter* vs. *exit*).
- The caller site: object reference and method name.
- The callee site: object reference and method name.

This structure is immediately implied by the semantic details of interception (recall Fig. 9). One can suspect easily that such a cache turns into a bottleneck. The large number of entries affects access time, the non-trivial structure of the keys (i.e., a 5-tuple) alike. We will now discuss ways to improve the situation. It turns out however that a drastic improvement can only be achieved by integrating a different technique: we have to go for a class-based scheme of interception; to be discussed in Sec. 4.3.

**Cache per MCI point** Instead of treating *dispatch* vs. *enter* vs. *exit* as part of a 5-tuple key, we can simply arrange for caches per point, each with a 4-tuple key then.

**Distributed cache** Instead of using a global cache, we can also distribute the cache on the caller or callee objects. Then, an inquiry also involves a cache which can be accessed via the caller / callee arguments that are part of the inquiry anyway. We can choose between per-caller caches or per-callee caches. By a per-caller (vs. per-callee)

cache, we mean that the cache that is associated with a given object maintains entries that are relevant for this object when serving as a caller (vs. a callee). In either case we get rid of another component of the key for caching. So we go from a 4-tuple to a 3-tuple. This is a major relief.

**Per-caller vs. per-callee caches** In theory, the per-caller and per-callee options are symmetric. In practice, the decision matters. For example, an argument in favour of per-callee caches is the following. MCI-like examples, as found in the AOP literature, tend to constrain more often the callee and not the caller. Then, a per-callee cache can easily shortcut for entries that are known to be generally applicable for all callers. This will drastically reduce the overall number of cache entries. Per-callee caches also seem to fit with potential virtual-machine support for MCI. That is, a method call dispatches on the callee anyway; so it will be natural to combine the virtual method table (VMT) look-up with MCI hashing on the callee.

**Fusion and vectorisation** Instead of treating the caller's and the callee's method names as different components of the key, we can use a combined identifier that is generated during source-code instrumentation. So we could go from a 3-tuple to a 2-tuple. We can also apply some form of vectorisation. That is, the method calls in a class can all be associated with a distinguished cache. These caches would need to be located on the caller site (because they relate to code locations of the caller). This scheme of vectorisation resolves the method name of the caller and the callee. So we obtain a cache with just a single component for the callee. As we see now, there are also arguments in favour of caller-site caches. (If we later go from object-based to class-based interception, the remaining component of the cache can be eliminated, too. This will eliminate the need of hashing completely.)

**Re-enabling garbage collection** Provisions are needed to harmonise caching and garbage collection. Firstly, the cache should not maintain proper references to caller and callee objects because otherwise these objects are never deallocated although they might be dead otherwise. Our actual scheme for source-code instrumentation adds an extra object-identifier attribute to each class. This attribute rather than the object reference itself is then used in the cache to identify caller and callee objects. Another problem concerns a distributed cache. Here, the MCI registry needs to maintain a container with *all* caches for the purpose of a reset operation. This again might prevent the caches from being garbage-collected even if the corresponding objects are effectively dead. One can use Java's weak references to deal with this problem (as of JDK 1.2). A language-independent, low-level technique is to associate time stamps with all caches. Before the MCI registry queries a cache it checks the master time stamp against the local time stamp to establish that the cache is not outdated.

## 4.2 Bypassing MCI inquiries

The overhead for MCI inquiries is relatively high. In particular, method calls that are not intercepted should preferably be executed much more directly. To see the problem,

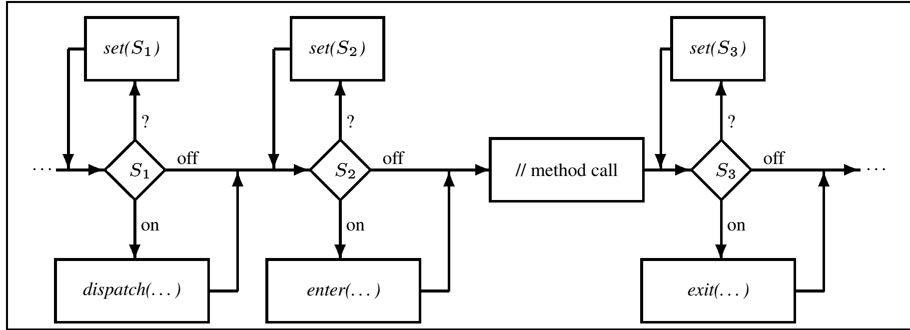



Figure 19: Bypassing MCI inquiries

we list all actions that are carried out even if no relevant superimposed functionality needs to be executed in the end:

- The method arguments must be placed in a hash table.
- Three inquiry messages have to be constructed and sent.
- The MCI registry or the cache is checked to be empty three times.
- Possibly modified method arguments have to be extracted from the hash table.

Under the assumption that many or most method calls are not intercepted, this overhead is clearly unacceptable. We use a bypassing technique to improve the situation.

**Switching logic** We assign a switch, i.e., a variable of the Java type `short`, to each of the points *dispatch*, *enter* and *exit* for each method call. If we encounter a method call for the first time, the corresponding switch will be uninitialised, and hence a proper MCI inquiry will be performed. If superimposed functionality was found, then the switch is set on, otherwise it is set off. If the method call is executed for the second time, the query can be immediately bypassed if the switch is off. In Fig. 19, this scheme is illustrated for the points *dispatch*, *enter* and *exit*. Bypassing can be naturally combined with caching. This is clearly attractive because bypassing only accelerates method calls that are known not to be intercepted whereas caching is needed to accelerate repeated inquiries for intercepted method calls.

In Fig. 20, we illustrate this refinement of source-code instrumentation for our running `fac`torial example. In fact, at the same time, the code snippet illustrates the use of a per-callee cache (see `_callee._cache`). The name `_inquiry_0` is the generated name for the switch variable associated with the recursive method call. The switch statement checks if a previous initialisation had already enabled bypassing (cf. `-1`). Otherwise the initialisation is performed (cf. `0`), or the inquiry is performed (cf. `1`). In the branch for initialisation, it is tested if a fall through to the case for performing the inquiry is needed, or if a bypass was attested (cf. the second `break`). The shown scheme can be further optimised to only use a single switch variable for all three points *dispatch*, *enter* and *exit*, and to perform only a single initialisation.

```
switch (this._inquiry_0) {

 case -1 : // Flag is off; so bypass inquiry
           break;

 case  0 : // Flag needs to be initialised
           if ((this._inquiry_0 =
              MCI_registry.test_dispatch(
               _callee._cache,
               this,"fac","integer",this._fac_sig,
               _callee,"fac","integer",_callee._fac_sig)
           ) != 1) break;

 case  1 : // Flag is on; so perform inquiry
           MCI_registry.dispatch(
            _callee._cache,
            this,"fac","integer",this._fac_sig,
            _callee,"fac","integer",_callee._fac_sig);

};
```

Figure 20: Switching logic at the dispatch point

**Switch consistency** Extra provisions are needed to ensure consistency of the switches. The available options are similar to what we discussed for caching. That is, one option is that the MCI registry maintains a container with objects that support *reset*. Another option is that the switching logic compares a local time stamp with a master time stamp.

## 4.3 Class-based interception

The pattern form "**object** *exp*", which can be used within MCI events, clearly reveals that the precision of interception can be object-based. That is, one can superimpose code on a per-caller, per-callee basis. This fine-grained control implies that the keys for caching would normally need to involve the caller and the callee. Using a distributed cache, we were able to get rid of either the caller or the callee in the key. We can radically optimise MCI inquiries if we sticked to class-based interception by means of removing the **object** ... form of a site pattern. In fact, MCI-like examples, as found in the AOP literature, tend to constrain the class but not the object. Object-based control can still be simulated by adding conditionals to the superimposed code blocks. The caller and callee objects could still be constrained in these conditionals by referring to the **caller** and **callee** descriptors. This trick can be incorporated into the compilation scheme by means of a semantics-preserving transformation. Then, the programmer can rely on object-based precision, which is however executed according to a more efficient, class-based regime.

```
1  class MyClassA {
2   ...
3   public void myMethA (...) {
4    ...
5   }
6  }
7
8  class MyClassB {
9
10  ...
11
12  public void myMethB1 (MyClassA A) {
13    MCI_aspect myAspect =
14      superimpose myMethB2() onto
15       dispatch: *->[object A];
16
17    ...
18  }
19
20  public void myMethB2 () {
21
22    ... // superimposed functionality
23
24  }
25 }
```

```
1  class MyClassA {
2   ...
3   public void myMethA (...) {
4    ...
5   }
6  }
7
8  class MyClassB {
9   [MyClassA myMciA = null;]
10  ...
11
12  public void myMethB1 (MyClassA A) {
13    MCI_aspect myAspect =
14      superimpose myMethB2() onto
15       dispatch: *->[descendant MyClassA];
16     [myMciA = A;]
17     ...
18  }
19
20  public void myMethB2 () {
21    [if (callee == myMciA) {]
22     ... // superimposed functionality
23    [}]
24  }
25 }
```

Figure 21: From object-based to class-based MCI

**From object-based to class-based MCI** In Fig. 21, we illustrate the transformation of an object-based superimposition so that class-based expressiveness is sufficient. In the object-based formulation on the left, there is a class `MyClassB` with a superimposition that intercepts method calls to a given object `A` of class `MyClassA` (or a subclass of it). The superimposed code block is placed in a method `myMethB2`. The class-based encoding is shown on the right of Fig. 21. The modified code fragments are boxed. Most notably, the MCI event is modified to constrain the class of the caller, only. Hence, we need an instance variable `myMciA` to keep track of the object `myMciA` the calls to which are to be intercepted; see the assignment "`myMciA = A;`". The superimposed functionality in the method `myMethB2` is wrapped by a condition to check if the `callee` at hand happens to coincide with `myMciA`.

**Caching revisited** The class-based regime improves caching and hashing as follows:

- Caches are now per-class if they were per-object before.
- The size of the cache is bound by the number of method calls in the source code.
- Cache hits are more likely because of the decreased cache precision.
- Vectorisation can be used to eliminate hashing completely as discussed in Sec. 4.1.

```
1  switch (this._inquiry_0) {
2
3   case -1 : // Flag is off; so bypass inquiry
4             break;
5
6   case  0 : // Flag needs to be initialised
7             if ((this._inquiry_0 =
8              MCI_registry.test_enter(
9                _callee._cache,
10               this,"fac","integer",this._fac_sig,
11               _callee,"fac","integer",this._fac_sig)
12            ) < 1) break;
13
14  case  1 : // Variable environment needed
15            _arg_1 = n-1;
16            _env = new Hashtable(1,1);
17            _env.put("n",new Integer(_arg_1));
18            MCI_registry.enter(
19              _callee._cache,
20              this,"fac","integer",this._fac_sig,
21              _callee,"fac","integer",_callee._fac_sig,
22              _env);
23            _arg_1 = (_env.get("n")).intValue();
24            break;
25
26  case  2 : // Variable environment not needed
27            MCI_registry.enter(
28              _callee._cache,
29              this,"fac","integer",this._fac_sig,
30              _callee,"fac","integer",_callee._fac_sig,
31              _env);
32 }
```

Figure 22: On-demand environment construction

## 4.4 Lazy MCI inquiries

So far, we applied the scheme of source-code instrumentation for method calls (recall Fig. 13) in an eager manner. That is, all actions of the instrumented method call were meant to be executed in the given order. One can now think of applying this scheme more lazily. In particular, a simple investigation reveals that the construction of the argument environment before an inquiry and the extraction of the possibly adapted arguments after the inquiry are very expensive. Construction only has to be performed if the superimposed functionality (if any) reads the arguments. Extraction is only needed in case the superimposed functionality (if any) modifies arguments. We use a refined switching logic to get to know from the registry if construction and ex-

traction are needed. This is illustrated in Fig. 22. For simplicity, we only consider the cases that both construction and extraction are performed, or neither of them. The figure shows the switch statement that handles the point *enter*. A four-valued logic is assumed now where an additional `case 2` leaves out the argument treatment.

### 4.5 Method wrappers

Our initial approach instruments each and every method call in the source code. The three points per method call require separate flags, switching logic, and others. Rather than instrumenting method calls directly, one can also wrap methods. Method calls are then trivially instrumented by calling the wrapper instead of the wrapped method itself. As a consequence, the code for instrumentation is only added once per method, which implies that the increase of code size will be reduced. A few more considerations are required to obtain a faithful model of method wrapping:

- We would like to ensure that all patterns for MCI events can still be used. This implies that the caller object needs to be passed to the method wrapper (unless we attempted to use low-level techniques for accessing the call stack).
- Using just a single wrapper for each method, we cannot support the point *dispatch* because the arguments will already be evaluated when control reaches the wrapper. One can however provide an extra wrapper per method just for the point *dispatch*. In this case, the instrumented method call will first call the extra wrapper, and then proceed with the normal method wrapper. This will increase code size only very little.
- When the switching logic was still associated with calls, this implied that the switches were specific to calls indeed. Now that the switching logic moves to the called methods, we either have to keep the variables for the switches on the caller site, or we adapt the switching logic to switch per called method instead. This is done in our implementation, which is defendable on the basis of the assumption that MCI events tend to be more concerned with callee than with caller sites.

This completes the discussion of refinements of the naive implementation of MCI.

## 5 Benchmarks for the MCI implementation

We will now assess our models by means of benchmarks.
These are the main questions that we want to address.

- What is the distributed fat for MCI?[5]
- Is there a prime model that can be clearly favoured over all other models?

[5]In general, the term 'distributed fat' stands for the run-time overhead caused by a language construct even if it is not used in an actual program. A prototypical example is multiple inheritance in a language like C++. The run-time efficiency of late binding slightly degrades if multiple inheritance is anticipated. Because of separate compilation it does not matter whether the program happens to use multiple inheritance or not.

- How does a method call slow down in various MCI scenarios?
- What are the contributions to run-time overhead?
- How does source-code instrumentation affect code size?
- How do the MCI models compare to static weaving approaches?
- How do the MCI models compare to other dynamic weaving approaches?

Other reports on the implementation of related constructs, most notably aspect-oriented expressiveness, tend to discuss such questions as well [9, 40, 7, 39]. Before we discuss our specific benchmarks, we will first characterise the benchmarking environment, and the variation points in programming with MCI.

**The benchmarking environment** All benchmarks are performed on an architecture with a Celeron 1100 Mhz with 384 MB RAM and running Linux with a 2.4.20 kernel, SuSE-Linux 8.2, and Java 1.4.2; more precisely: Java(TM) 2 Runtime Environment, Standard Edition (build 1.4.2-b28) Java HotSpot(TM) Client VM (build 1.4.2-b28, interpreted mode).

**Variation points** One way to measure performance of an MCI language implementation is to use some 'typical' programs that involve MCI. At this stage, there are not many programs available that are written in the style supported by the MCI approach, but we could try to find some typical programs in the AOP context, and maybe adapt them. We have decided to opt for a complementary approach instead: we measure the performance by exploring the variation points for MCI code. Here is an inventory of natural variation points, which are explored in the sequel to some extent:

1. Number of classes, methods, method calls.
2. Number of superimpositions.
3. Complexity of MCI events.
4. Ratio intercepted to non-intercepted calls.
5. Ratio method calls to remaining program actions.
6. Ratio superimposed code to base code.
7. Relevance of caller/callee for a given superimposition.
8. Relevance of environment for a given MCI superimposition.
9. Class-based vs. object-based interception.
10. Frequency of the same method call.
11. Update frequency for MCI registry.

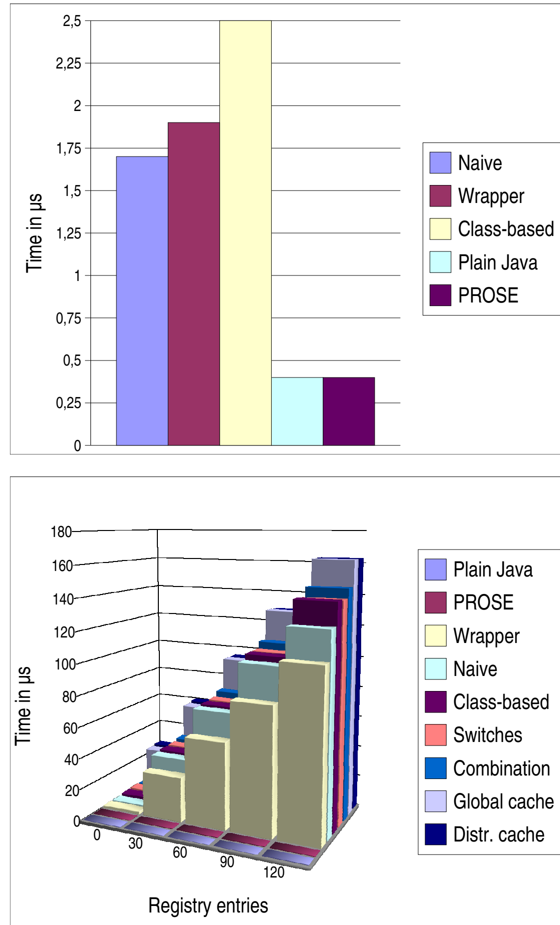



Figure 23: First execution without interception

## 5.1 Prime MCI model

Given the naive model and the various proposals for refinement, we want to identify a prime model. Note that some optimisations can be used jointly, which increases the number of options to consider. We identified the following 'interesting' models:

**naive** the naive model as of Sec. 3.

**global cache** as of Sec. 4.1.

**distributed cache** as of Sec. 4.1.

**switches** for bypassing as of Sec. 4.2.

**combination** distributed cache + bypassing.

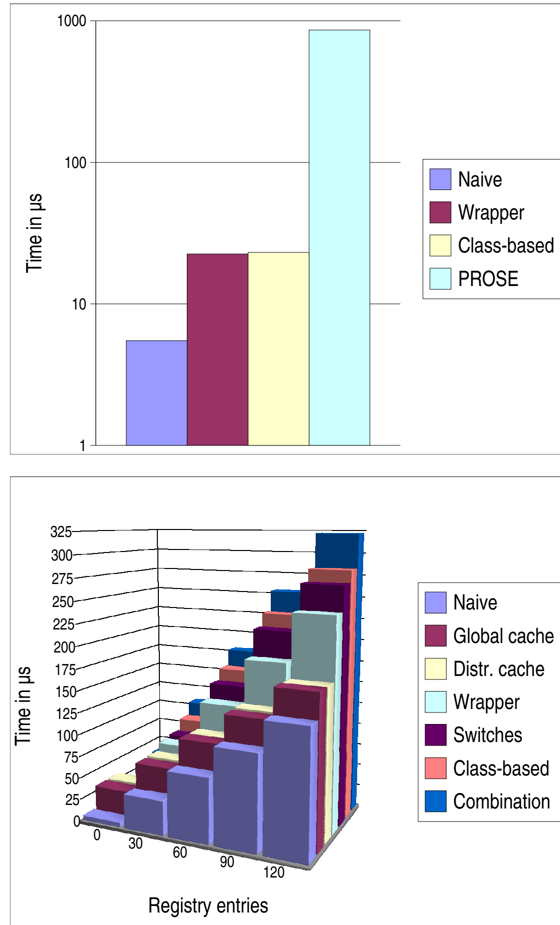



Figure 24: First execution with interception

**class-based** MCI as of Sec. 4.3 + distributed cache + bypassing.

**wrapper** class-based model with callee-site wrapper methods as of Sec. 4.5.

Our measurements are based on the following rationale:

- To illustrate the virtue of caching and bypassing, we distinguish two cases as follows: a) an observed method call is executed for the *first* time; b) a *repeated* execution is encountered.
- To address the issue of distributed fat, we distinguish two cases as follows: a given method call happens to be intercepted or not. (One would wish that nonintercepted calls are executed with very little overhead. For intercepted calls, some overhead would be acceptable if we assume that, in a complex software system, most method calls are not intercepted.)

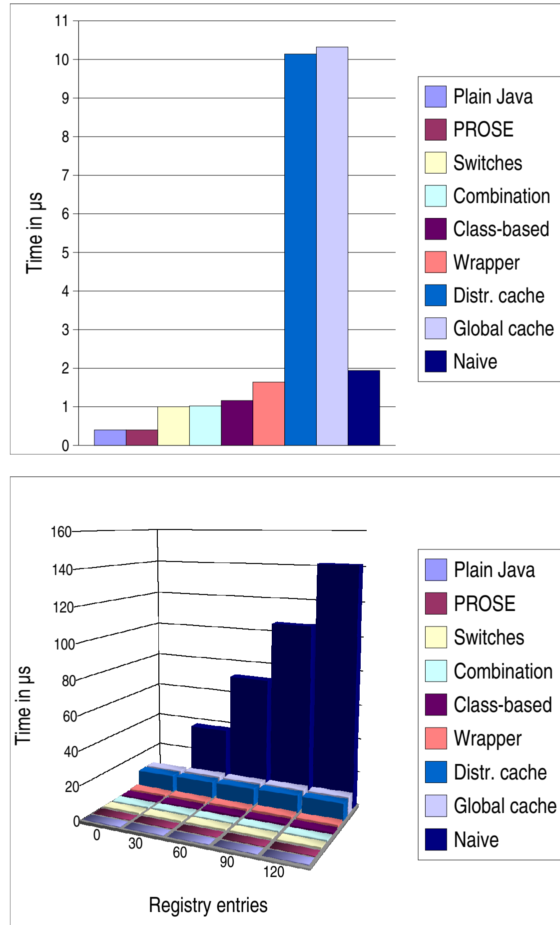



Figure 25: Repeated execution without interception

- To demonstrate the impact of the MCI registry's size. we vary the number of entries in the MCI registry; we use the range 0, . . . , 120.
- Initially, we favour micro measurements, i.e., we want to measure costs of a single method call. We will later see benchmarks for complete program executions.

The benchmark results are listed in Fig. 23 through Fig. 26. (For the sake of the comparison of MCI with another run-time adaptation approach, we also include PROSE (as of [40]) in some of the figures, but we will talk about this comparison later.) We can draw the following conclusions from this data:

- The moment to pay for MCI is really when a method call is performed for the first time. This is because the MCI look-up happens precisely then. To see the overhead, we look at Fig. 23 (and maybe at Fig. 24, too). We note that the cost

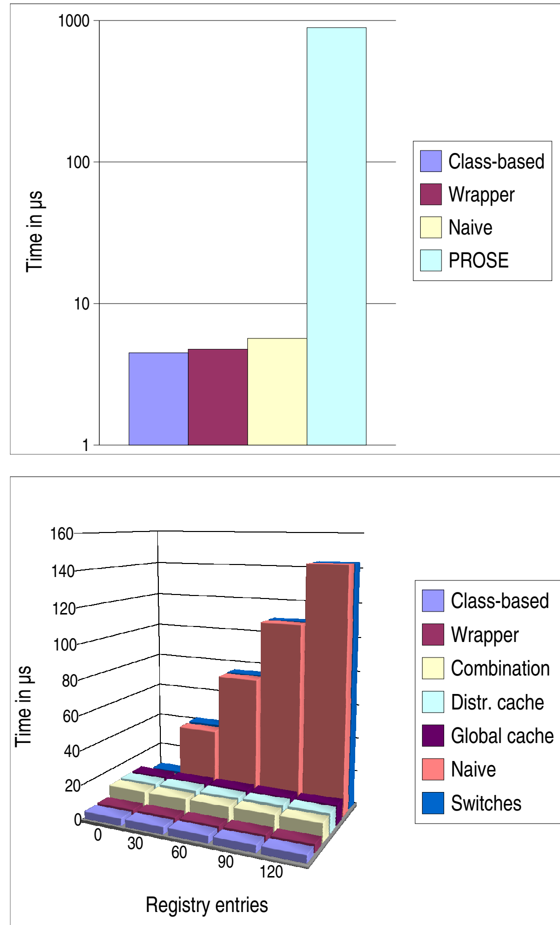



Figure 26: Repeated execution with interception

for a plain method call in Java is $0.4\mu$s. Then, for an empty registry, the MCI-enabled method call for the naive model is a factor 4 slower than in plain Java.

- All our optimisations cannot help in any way with the first execution of a method call. In fact, the optimised models in Fig. 23 (and Fig. 24) are slightly slower than the naive model because of the extra logic. We should emphasise that, in the class-based model, the overhead for the first execution will only be paid once per method call in the source code.
- The overhead for the first execution is largely independent of the fact if the method is intercepted or not because all MCI registry entries need to be scanned anyway. We can see this by comparing Fig. 23 and Fig. 24 for the small numbers of registered entries. The above factor 4 increases linearly with the number of MCI entries to be checked.

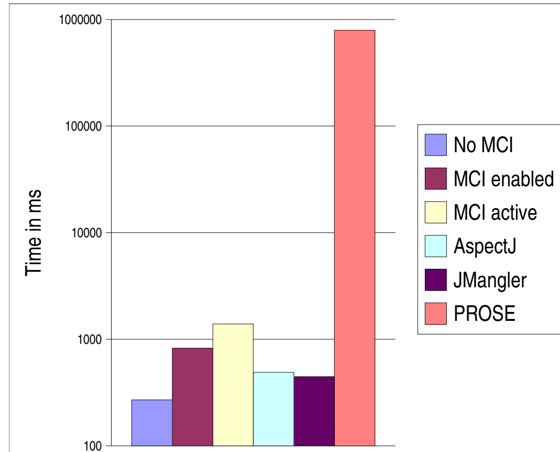



Figure 27: Comparison with other technology

- The virtue of our optimisation techniques becomes apparent when we start to assess the figures for repeated execution. According to Fig. 25, a nonintercepted call is only a factor 2.5 slower than in plain Java — at least for some MCI models. This is mainly achieved by the bypassing technique. In fact, we see that all models that do not attempt bypassing are hard to defend.
- According to Fig. 26, an intercepted call is at least by a factor 11.5 slower than a plain Java call. Caching becomes important with an increasing number of entries in the MCI registry.

We decide to declare the *class-based* model as the winner, but the wrapper model is another candidate because of the reduced code-size increase that can be expected from it. In terms of run times, it is very close to the class-based model. In fact, the wrapper model is sometimes faster than the class-based model because the initialisation of switches does not look at the caller pattern of an MCI event. This loss of precision could be problematic for some MCI programs however.

## 5.2 Comparison with static weaving technology

From an AOP perspective, MCI provides expressiveness for dynamic weaving — even though the source code is generally prepared during compile time. Dynamic weaving is potentially less efficient than static weaving for obvious reasons. To measure this in some way, we compare our approach with AspectJ [46] and JMangler [6]. AspectJ is a true compile-time weaving approach, while JMangler is a load-time approach. (We removed the load-time overhead in our measurements to only show times for the actual execution of weaved programs.) For our measurements we use AspectJ 1.0.6 and JMangler 2.0. The approaches are compared in Fig. 27 using a sorting program with a superimposition for counting all method calls as a benchmark. A list with 1000

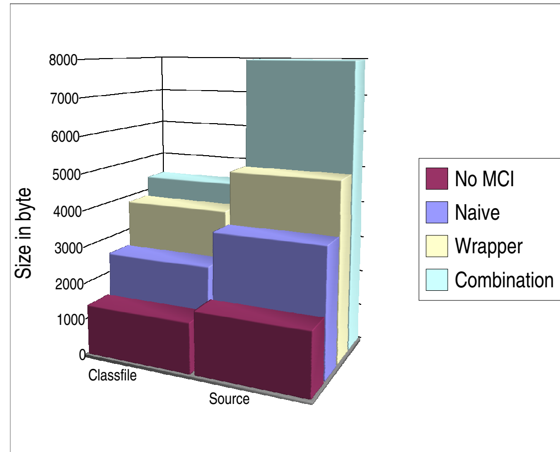



Figure 28: Code size

elements is sorted in terms of 504503 method calls. The diagram suggests that our prime MCI model is factor 3 slower than AspectJ. This is a kind of worst-case scenario because all calls are intercepted. We did not run arbitrary Java or AspectJ benchmark suites because our implementation does not cover full Java, neither does the present form of MCI cover full AspectJ.

### 5.3 Comparison with dynamic weaving technology

In the static arena, there are several further candidates that could be integrated in our comparison, but we view AspectJ and JMangler as sufficiently indicative here. In the dynamic arena, there are few candidates that could easily be compared with MCI. For example, there is JAC [36, 13] whose run-time adaptation capabilities are however rather limited. We have compared MCI with PROSE as of [40], which employs the Java Virtual Machine Debugger Interface for run-time weaving. For our measurements we used PROSE 0.17.1-beta. (We must mention that there is a recent implementation of PROSE which uses a completely different architecture. It is based on the Jikes VM; see related work on native support.) The comparison prime MCI model vs. PROSE provides the following facts:

- For the sorting benchmark, PROSE exhibits a factor 570 slow-down compared to the prime MCI model.
- The micro measurements for intercepted method calls suggest a smaller but still considerable factor 200; see Fig. 24 and Fig. 26. PROSE seems to be stressed by larger number of objects and calls.
- PROSE does not imply any overhead for nonintercepted method calls. In fact, the PROSE data in Fig. 23 and Fig. 25 are equivalent to plain Java method calls.

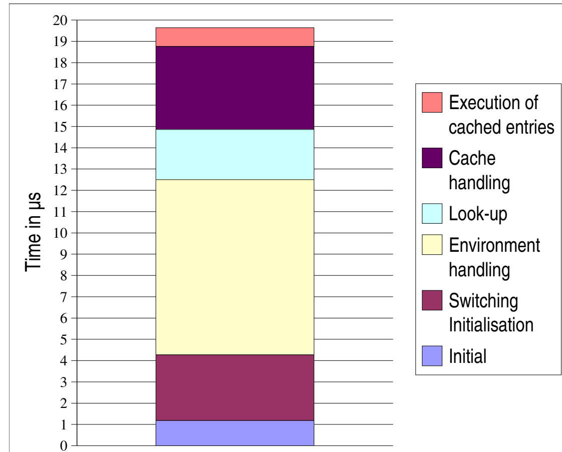



Figure 29: Method call chart

## 5.4 Size of instrumented code

The increase in code size due to instrumentation, though linear, is considerable; see Fig. 28. Depending on the model, we might have a factor 3.5 increase for class files and a factor 6 increase for source files. Obviously, the more advanced a model, the higher the code increase. The achievement of method wrapping is quite visible in Fig. 28. That is, the amount of code sharing achieved by wrapping methods cuts the code size by half. This holds despite the fact that the present wrapping model builds on top of the class-based model which subsumes caching and bypassing. (We should mention that the present wrapper models ignores the caller in its switching logic, but this can be neglected as for code size.) Regarding further decrease of code size, one could attempt to share code even between wrappers of different methods or classes.

## 5.5 Method call chart

In Fig. 29, we analyse the ingredients of the factor for slowing down method calls. To this end, we determine the contributing factors for the various steps involved in an MCI inquiry according to our scheme of source-code instrumentation. We consider a single intercepted method call with one argument of type `int`. The body of this method and the superimposed code block are NO-OPs. The first beam is about the plain method call and the switching logic but pre-initialised to skip all inquiries. (This is comparable to the repeated encounter of a method call where bypassing of MCI inquiries is applicable. This is as fast as we can get with MCI.) The next beam adds the initialisation of the switches at the *enter* point. The next beam shows the time spent on constructing the environment with one argument. Then, registry look-up is added, and so on. In the diagram, we did not apply on-demand environment construction. Thereby, we clearly illustrate the costs of environment construction. This cost can be eliminated whenever the superimposed functionality does not access the arguments; recall Sec. 4.4.

This completes the discussion of benchmarks for the implementation of MCI.

# 6 Related work

**Superimpositions for distributed systems** Our notion of superimposition of functionality onto method calls is inspired by superimposition as defined elsewhere for distributed systems of communicating processes [11, 25, 8, 42], say for parallel program design. This link is also explainable with reference to Aspect-Oriented Programming (AOP). Just as in AOP, a superimposition for distributed systems is orthogonal to the usual breakdown of modules. A typical example of a superimposition is repeated checking of whether deadlock has occurred within a system of processes. In [26], Katz and Gil argue that AOP language design and formal foundations could benefit from the body of knowledge about superimposition for distributed systems. In a way, our operational semantics substantiates this link: the semantics describes quite explicitly how the execution of normal method calls is 'interleaved' with added functionality at distinguished join points. This is just as in a distributed system, where the superimposed actions are interleaved with the normal execution of the processes, which communicate via channels (as opposed to message sends).

**Formal models of aspect-oriented programming** MCI contributes to the foundations of run-time evolution and AOP while staying close to normal OO languages. There are other contributions to the AOP foundations, which do not directly relate to an object-oriented base language [5, 17]. The model by Douence, Motelet and Südholt [17] can be seen as an abstract, more language-independent form of interception / superimposition facilities. The model is based on execution monitors for the events corresponding to the points of interest along the execution of a program. Points of interest that relate to one another (in the sense of an aspect) are denoted by patterns of events to be matched. In [44], Wand et al. address the semantics of specific expressiveness in the Java-based AOP language AspectJ, namely dynamic join points. Still this semantics is not built on top of an object-oriented base semantics, and issues of static typing are not addressed. There is a large body of related work on the foundations of reflection. Notions similar to MCI and corresponding idioms for implementation were studied in the context of Dalang / Kava [45], but without considering a designated operational semantics for higher-level constructs like MCI.

**Component adaptation and evolution** MCI provides means for run-time system adaptation that appeal to the obliviousness and quantification principles of AOP as identified by Filman and Friedman [19]. There is a considerable body of work on other forms of system adaptation that are more related to anticipated evolution along explicit join points, e.g., the work by Andrade, Bosch, Fiadeiro [10, 4, 21]. That is, extra functionality is superimposed onto interfaces of otherwise black-box components while enumerating 'connectors' one-by-one. Some concrete forms of such adaptation scenarios are generalised adaptors (in the sense of the design pattern of this name), generalised wrappers, and superimposition in the sense configurable architectural connectors. All this work depends on special object models, generative tool support and

use of design patterns for deployment, whereas our form of MCI is ready for tight integration with OO base languages. There are several characteristics of the above-mentioned work that propose important extensions of MCI, e.g., the use of guards, or pre- and postconditions for a model of evolution. Also, MCI could maybe cover forms of adaptation that affect interfaces. Finally, superimpositions that require the coordination of multiple partners will be a challenging generalisation of MCI.

**Metaobject protocol** Our aspiration regarding the semantical and implementational development of MCI is to substantiate the status of MCI to be regarded as an immediate language construct. In a language with a Metaobject protocol (MOP; [29, 28]), say full reflection as opposed to Java's 'read-only' reflection, MCI can be accomplished as a higher layer on top of the MOP. For example, this is evident in the Smalltalk-based framework AspectS by Hirschfeld [22]. A style of reflective programming that leads to efficient and type-safe programs while still being sufficiently flexible is an ongoing research theme [12, 33, 45]. The composition filter approach to AOP by Aksit et al. [3] also relies on a general form of reflection. Composition filters provide a higher layer such that object interactions can be reified, matched and filtered in various ways. Our form of MCI is strictly more restrictive.

**Technology for instrumentation** Source-code instrumentation can be realised using means for compile-time reflection — as supported by Shiba's et al. OpenJava [43]. The corresponding transformation rule is then encoded as reflective code on method bodies (for either wrapping method calls or entire bodies of callees). We have experimented with OpenJava to an extent that we are confident regarding its suitability for an implementation of MCI that covers full Java. Instead of instrumenting source code, one can also consider byte-code engineering as supported by JMangler, BCA, or Javassist [6, 27, 14]. Byte-code transformation can either happen at class-load time, or offline. Java's byte code does not provide all the information from the source code, in particular the names of method parameters are missing. Hence, the corresponding MCI pattern cannot be supported like this. Byte-code transformation is not very convenient for expressing the non-trivial transformation rule for instrumentation, which involves helper variables and others. Additional opportunities arise from using technology for replacing (Java) classes at run-time of an application — as addressed by Gustavsson's et al. JDrums [23]. Then, the instrumentation can be delayed, or superimposition can be performed on-the-fly.

**Method wrappers** Baker's and Hsieh's Handi-Wrap [9] provides a particular implementation of method wrapping in Java, which is similarly related to AOP as our approach. While we instrument calls to query the registry, Handi-Wrap installs (possibly cascaded) method wrappers in a designated field on the callee site. These fields are filled at weaving time, and each method body uses a simple scheme to iterate over the wrappers. This approach requires loading classes from the class-path when an aspect is added. The Handi-Wrap paper provides a general overview on wrapping techniques.

**Native support** We think that the ultimate approach to run-time system adaptation would indeed be complemented by native run-time support either at the level of a virtual machine (VM) or a just-in-time (JIT) compiler. A promising approach is reported in [39] by Popovici, Alonso and Gross. JIT compilation and the Jikes VM are employed in this case to construct a new version of the aspect-oriented framework PROSE. Compared to the original version, which relied on the JVM Debugger Interface [40], the new version achieves a performance improvement by a factor 16–151.

# 7 Concluding remarks

We have presented simple constructs for the run-time adaptation of object-oriented programs by means of method-call interception (MCI). We have described the static and dynamic semantics of MCI. We have reported in detail on a semantics-directed approach to language implementation, where we use source-code instrumentation to compile away MCI constructs. This approach applies to statically typed, class-based, compiled languages such as Java. The given implementation maintains static type safety and separate compilation. In terms of efficiency, the approach gets close to static weaving approaches while providing the added value of run-time adaptation capabilities. We share the widely established experience that source-code instrumentation is quite convenient for the exploration of new language constructs — when compared to the ultimate design of accordingly enhanced virtual machines or just-in-time compilers. Our development pinpointed refinements of a naive implementational model on the basis of variation points for OO code that involves MCI. We have used benchmarks to demonstrate the overall performance of the MCI models and the benefits of the refinements.

# References


[1] *Proc. 1st Int'l Conference on Aspect-Oriented Software Development (AOSD'02)*. ACM Press, Apr. 2002.

[2] F. Akkawi, A. Bader, and T. Elrad. Dynamic Weaving for Building Reconfigurable Software Systems. In *Proc. OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, 2001.

[3] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa. Abstracting Object Interactions Using Composition Filters. In R. Guerraoui, O. Nierstrasz, and M. Riveill, editors, *Proc. ECOOP'93 Workshop on Object-Based Distributed Programming*, volume 791 of *LNCS*, pages 152–184. Springer-Verlag, 1994.

[4] L. Andrade and J. Fiadeiro. Coordination: The Evolutionary Dimension. In W. Pree, editor, *Proc. Technology of Object-Oriented Languages and Systems (TOOLS'01)*, pages 136–147. IEEE Computer Society Press, 2001.

[5] J. H. Andrews. Process-algebraic foundations of aspect-oriented programming. In *Proc. Third Int'l Conference on Metalevel Architectures and Separation of*

*Crosscutting Concerns (Reflection 2001)*, volume 2192 of *LNCS*, pages 187–209. Springer-Verlag, Sept. 2001.

[6] M. Austermann. JMangler Homgepage, 2002. `http://javalab.cs.uni-bonn.de/research/jmangler/index.html`.

[7] E. Avdi., M. Mernik, M. M. Lenic, and V. Zumer. Experimental Aspect-Oriented Language – AspectCool. In *Applied Computing 2002: Proc. (17th ACM) Symposium on applied computing*, Mar.11–14 2002.

[8] R. Back and K. Sere. Superposition Refinement of Reactive Systems. *Formal Aspects of Computing*, 8(3):324–346, 1996.

[9] J. Baker and W. Hsieh. Runtime aspect weaving through metaprogramming. In ACM [1], pages 86–95.

[10] J. Bosch. Superimposition: A component adaptation technique. *Information and Software Technology*, 41(5):257–273, 25 Mar. 1999.

[11] L. Bougé and N. Francez. A compositional approach to superimposition. In ACM, editor, *Proc. Conference on Principles of programming languages (POPL'88), January 13–15, 1988, San Diego, CA*, pages 240–249. ACM Press, 1988.

[12] S. Brandt and R. W. Schmidt. The Design of a Meta-Level Architecture for the BETA Language. In *Proc. META '95: Workshop on Advances in Metaobject Protocols and Reflection at ECOOP'95*, Aug. 1995.

[13] JAC - Java Aspect Components Site © CEDRIC Laboratory, 2002. http://jac.aopsys.com.

[14] S. Chiba. Load-Time Structural Reflection in Java. In *Proc. ECOOP'00—Object-Oriented Programming*, volume 1850 of *LNCS*, pages 313–336. Springer-Verlag, 2000.

[15] J. de Oliveira Guimarães. Reflection for statically typed languages. In E. Jul, editor, *ECOOP '98—Object-Oriented Programming*, volume 1445 of *Lecture Notes in Computer Science*, pages 440–461. Springer, 1998.

[16] T. Despeyroux. TYPOL: A formalism to implement natural semantics. Technical report 94, INRIA, Mar. 1988.

[17] R. Douence, O. Motelet, and M. Südholt. A formal definition of crosscuts. In *Proc. Third Int'l Conference on Metalevel Architectures and Separation of Crosscutting Concerns (Reflection 2001)*, volume 2192 of *LNCS*, pages 170–186. Springer-Verlag, Sept. 2001.

[18] T. Elrad, R. E. Filman, and A. Bader. Aspect-oriented programming: Introduction. *CACM*, 44(10):29–32, Oct. 2001. An introduction to the CACM special issue on AOP.

[19] R. Filman and D. P. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Proc. OOPSLA 2000 Workshop on Advanced Separation of Concerns*, Oct. 2000.

[20] M. Golm and J. Kleinöder. Jumping to the Meta Level: Behavioral Reflection Can Be Fast and Flexible. In P. Cointe, editor, *Proc. Reflection'99*, volume 1616 of *LNCS*, pages 22–39. Springer-Verlag, 1999.

[21] J. Gouveia, G. Koutsoukos, L. Andrade, and J. Fiadeiro. Tool Support for Coordination-Based Software Evolution. In W. Pree, editor, *Proc. Technology of Object-Oriented Languages and Systems (TOOLS'01)*, pages 184–196. IEEE Computer Society Press, 2001.

[22] R. Hirschfeld. AspectS – Aspect-Oriented Programming with Squeak. In M. Aksit, M. Mezini, and R. Unland, editors, *Objects, Components, Architectures, Services, and Applications for a Networked World*, volume 2591 of *LNCS*, pages 216–232. Springer-Verlag, 2003.

[23] JDrums, Java Distributed Run-time Updating Management System, 2003. `http://www.ida.liu.se/~jengu/jdrums/`.

[24] G. Kahn. Natural Semantics. In *4th Annual Symposium on Theoretical Aspects of Computer Science*, volume 247 of *LNCS*, pages 22–39. Springer-Verlag, 19–21 Feb. 1987.

[25] S. Katz. A superimposition control construct for distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(2):337–356, Apr. 1993.

[26] S. Katz and Y. Gil. Aspects and superimpositions. In *Proc. Int'l Workshop on Aspect-Oriented Programming (ECOOP'99)*, June 1999.

[27] R. Keller and U. Hölzle. Binary component adaptation. In E. Jul, editor, *ECOOP '98—Object-Oriented Programming*, volume 1445 of *LNCS*, pages 307–329. Springer, 1998.

[28] G. Kiczales, J. Ashley, L. Rodriguez, A. Vahdat, and D. Bobrow. *Metaobject Protocols: Why We Want Them and What Else They Can Do*, pages 101–118. The MIT Press, 1993.

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

[30] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *ECOOP 2001 — Object-Oriented Programming 15th European Conference*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer-Verlag, June 2001.

[31] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Aksit and S. Matsuoka, editors, *Proc. ECOOP'97—Object-Oriented Programming, 11th European Confernce*, volume 1241 of *LNCS*, pages 220–242. Springer-Verlag, 9–13 June 1997.

[32] I. Kiselev. *Aspect-Oriented Programming with AspectJ*. Sams Publishing, 2003.

[33] J. Kleinoeder and M. Golm. MetaJava: An efficient run-time meta architecture for Java. In L.-F. Cabrera and N. Islam, editors, *Proc. Fifth Int'l Workshop on Object-Orientation in Operating Systems*, pages 54–61. IEEE Computer Society Press, 1996.

[34] R. Lämmel. A Semantical Approach to Method-Call Interception. In ACM [1], pages 41–55.

[35] P. Lee. *Realistic Compiler Generation*. Foundations of Computing Series. MIT Press, 1989.

[36] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. JAC: A Flexible Framework for AOP in Java. In *Reflection'01*, volume 2192 of *LNCS*, pages 1–24. Springer-Verlag, Sept. 2001.

[37] M. Pinto, L. Fuentes, M. Fayad, and J. Troya. Separation of Coordination in a Dynamic Aspect Oriented Framework. In ACM [1], pages 134–140.

[38] U. F. Pleban. Compiler prototyping using formal semantics. In *Proc. SIGPLAN '84 Symposium on Compiler Construction*, pages 94–105. ACM, ACM, 1984.

[39] A. Popovici, G. Alonso, and T. Gross. Just-in-time aspects: efficient dynamic weaving for java. In *Proc. 2nd Int'l Conference on Aspect-oriented software development*, pages 100–109. ACM Press, 2003.

[40] A. Popovici, T. Gross, and G. Alonso. Dynamic Weaving for Aspect Oriented Programming. In ACM [1], pages 141–147.

[41] The Rule Evolution Kit, version 0.42, 15 Oct. 2003. `http://www.cs.vu.nl/rek/`.

[42] M. Sihman and S. Katz. A calculus of superimpositions for distributed systems. In ACM [1], pages 28–40.

[43] M. Tatsubori, S. Chiba, M.-O. Killijian, and K. Itano. OpenJava: A Class-Based Macro System for Java. In W. Cazzola, R. Stroud, and F. Tisato, editors, *Reflection and Software Engineering*, volume 1826 of *LNCS*. Springer-Verlag, 2000.

[44] M. Wand, G. Kiczales, and C. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. In G. T. Leavens and R. Cytron, editors, *FOAL 2002 Proceedings: Foundations of Aspect-Oriented Languages Workshop at AOSD 2002*, pages 1–8, Apr. 2002.

[45] I. Welch and R. Stroud. From Dalang to Kava — the Evolution of a Reflective Java Extension. In P. Cointe, editor, *Proc. Reflection'99*, volume 1616 of *LNCS*, pages 2–21. Springer-Verlag, 1999.

[46] AspectJ.org Site © XEROX Corporation, 2002. http://aspectj.org.