IConJ 0.1: A Proof-of-Concept Tool for Applying Implicit Context to Java Software

Robert J. Walker
Department of Computer Science
University of Calgary
rwalker@cpsc.ucalgary.ca

Technical Report 2004-757-22

Abstract

Implicit context is a model for the generative adaptation of software modules that can be declaratively incomplete or inconsistent with other modules in a system. This technical report describes the syntax and semantics of a proof-of-concept tool (*IConJ 0.1*) for the application of the implicit context model to software written in the Java language.

1 Introduction

The model of implicit context is based upon the premise that, during software evolution, software reuse, and even "greenfield" software development, software modules often end up with dependencies on the system around them that are not satisfiable by the system as it is. Such dependencies can take the form of references to external type names, external interfaces, and external protocols. Rather than invasively modify such modules, it can be preferable to externally adapt them to their actual context of operation. Such adaptation can require the treatment of multiple, apparently identical type references as being to different types, or multiple, apparently different type references as being to the same type. Method calls can be replaced with other method calls, requiring stateful decisions. Full protocols can be transplanted with alternatives.

This technical report describes a proof-of-concept tool for the application of the implicit context model to software written in the Java language [Gosling *et al.*, 2000]. Adaptive transformations are explicitly declared in constructs called *boundary maps*. IConJ 0.1 combines Java source classes with particular boundary maps in order to adapt them. As such, IConJ 0.1 is a tool for the generative adaptation of Java source classes.

IConJ 0.1 is unusual in three key features. (1) It permits the input source classes to be *inconsistent*, prior to adaptation. That is, the constraints placed upon the context of operation by one source class may conflict with the constraints placed upon it by others. For example, an attempt to instantiate a class might be made, while that class is actually declared to be abstract. It is the task of the developer to use IConJ to eliminate such inconsistencies at composition time. (2) It permits the input source classes to be *declaratively incomplete*, prior to adaptation. That is, a reference may not resolve to any declarations, even ignoring questions of ambiguity and inconsistency. It is the task of the developer to use IConJ to bind such unbound references. (3) It permits stateful adaptation based upon the past *communication history* of the system. This allows the developer to provide more intentional declarations of protocols, rather than using poorly localized implementations.

IConJ 0.1 operates as a preprocessor for Java source code. It supports boundaries only around individual classes and methods, largely because these declarations are easily named in Java source; extending the kinds of boundaries remains a task for later versions. To support interception and alteration of communication at these boundaries, IConJ 0.1 provides a construct called a *boundary map*; system composers explicitly create boundary maps to perform contextual dispatch in order to compose inconsistent source fragments. Boundary maps are declared in files that are separate from the Java source code. System composers also explicitly declare, through *tool directives*, to which boundary or boundaries a given boundary map is to be attached.

We begin with a brief overview of the model of implicit context, in Section 2. Section 3 motivates the syntax and operation of boundary maps through a simple example. The general structure of IConJ 0.1 is described in Section 4 along with an overview of its usage, leaving the non-trivial details of its algorithms to the later sections of this report. The features of boundary maps as provided by IConJ 0.1 are discussed in Section 5. Tool directives are described in Section 6, and the sequential application of boundary maps is considered in Section 7. Finally, Section 8 describes some of the implementation issues of the tool. The appendix lists the complete application programming interface to communication history, and a detailed example of the inputs to and resulting outputs from IConJ 0.1.

2 A Brief Overview of the Model of Implicit Context

The model of implicit context consists of three concepts: *boundaries* between inconsistent source fragments (or classes or modules, etc.), *contextual dispatch* which is used to alter communications, and *communication history* which is used to retrieve previous state when performing contextual dispatch.

Contextual dispatch is a generalization of object-oriented dispatch. Contextual dispatch may be performed on communications as they cross a boundary between inconsistent source fragments. Rather than selecting the implementation of a method on the basis of the run-time type of a given object, we select the method or methods to execute on the basis of the context in which the communication occurs—both static properties and dynamic properties of the system. Contextual dispatch can be used to fill in concrete details in an abstract service request: a message can be rerouted to a concrete implementor of the service, with additional parameters filled-in.

Trouble can arise when the source of such additional parameters is not immediately obvious. Communication history can be used in these situations. Communication history is, conceptually, a record of all communication that has occurred in the system, including all arguments passed and values returned. Through (pseudo-)queries on communication history, we can retrieve state information of interest: the last passed argument of a particular type, for example. Communication history queries have the advantage that they can be expressed to minimize our assumptions about the system. Thus, communication history allows us to reduce the coupling in our systems, easing software evolution and reuse.

Further details of implicit context can be found elsewhere [Walker and Murphy, 2000; Walker, 2003]. Revised publications are underway and should appear in the near future.

3 A Simple Example

To examine the major features of IConJ 0.1, consider the following example that uses implicit context. We have a simple class, called `CompoundWidget`, for creating a particular kind of compound widget comprising a button and a scroll bar. Shown below is part of the declaration of `CompoundWidget`, including the declarations for two fields that hold its button and scroll bar objects and a constructor declaration.

```
public class CompoundWidget {
  private Button button;
```

```
  private ScrollBar scrollBar;

  public CompoundWidget() {
    button = new Button();
    scrollBar = new ScrollBar();
    // Perform other setup
  }

  // Other methods
}
```

This constructor instantiates the `Button` and `ScrollBar` classes, and stores each resulting object in the appropriate field of the newly-created instance of `CompoundWidget`. This source code would occur in a file named `CompoundWidget.java`.

Now, imagine that we have different versions of `Button` and `ScrollBar` for different platforms, such as a version operating with the Motif library to run on Unix systems and a version to operate on Microsoft Windows systems. We decide to use the Abstract Factory design pattern [Gamma *et al.*, 1994] to isolate these platform-specific differences from the rest of our system. The class structure at which we arrive is shown in Figure 1. Here, `Button` and `ScrollBar` are abstract base classes, each specialized into platform-specific versions. There is also a hierarchy of widget factories, consisting of an abstract base class called `WidgetFactory` and platform-specific subclasses. Each platform-specific widget factory creates instances of the corresponding platform-specific versions of `Button` and `ScrollBar`. Clients can ignore platform differences by possessing an instance of either subclass of `WidgetFactory` but treating it as being of type `WidgetFactory`. Calling the methods on this instance causes the appropriate instances of `Button` or `ScrollBar` to be created polymorphically. One part of the system must still create the appropriate widget factory and pass it to the client, and thus, this part will be aware of which platform is in use.

The developer of this system wishes to allow the interior of the `CompoundWidget` module to ignore the added complexity of Abstract Factory, while having the system as a whole utilize these platform-specific subclasses. From the perspective of the system as a whole, the declaration of `CompoundWidget` should effectively be the following:

```
public class CompoundWidget {
  private Button button;
  private ScrollBar scrollBar;

  public CompoundWidget(WidgetFactory fact) {
    button = fact.createButton();
    scrollBar = fact.createScrollBar();
    // Perform other setup
  }

  // Other methods
}
```

The differences are that an instance of `WidgetFactory` should be passed to the constructor, and that this instance should be used to build the individual pieces of the `CompoundWidget`. In fact, since `Button` and `ScrollBar` are abstract classes in the view of the system as a whole, they are not directly instantiable.

To permit these differences, we begin by considering there to be a boundary around `CompoundWidget` between these independent world views. Next, we must specify contextual dispatch at this boundary to resolve the conflict between the world views. The composer specifies three boundary maps to do this:

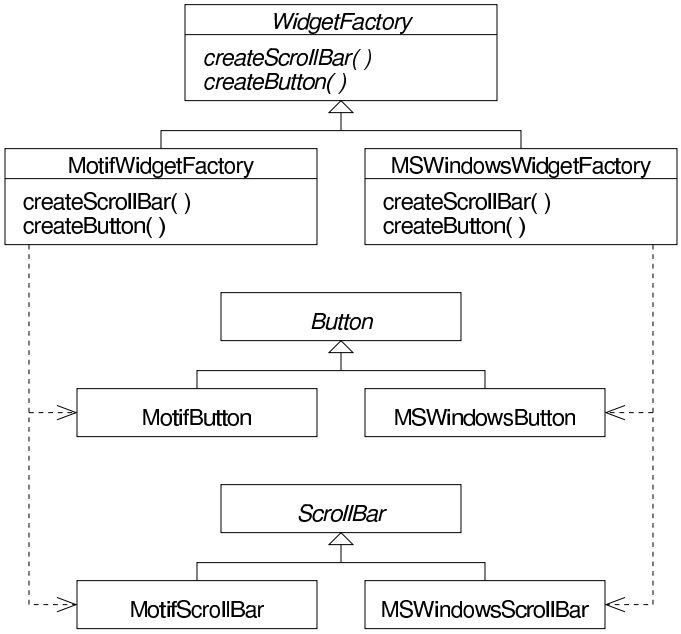


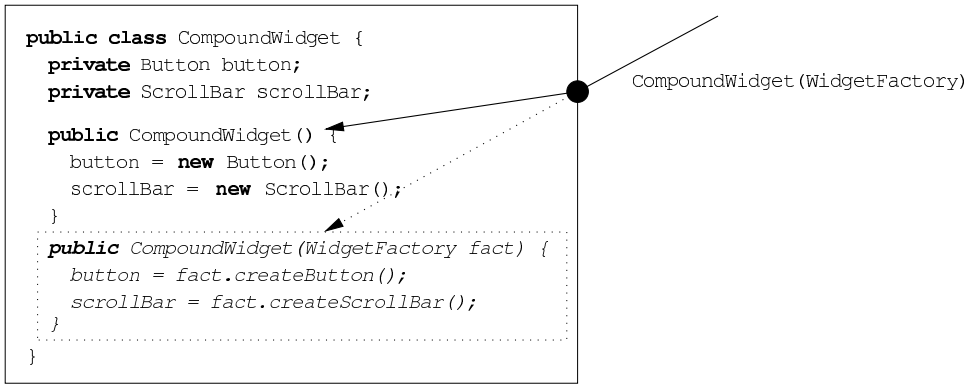
Figure 1: Using the Abstract Factory design pattern to isolate platform-specific dependencies. We have separate subclasses of `Button` and `ScrollBar` for both Motif and Microsoft Windows systems.

one that translates incoming attempts at constructing `CompoundWidget` instances parameterized by a `WidgetFactory` object; one that translates outgoing attempts at instantiating `Button`; and one that translates outgoing attempts at instantiating `ScrollBar`. The operation of these boundary maps is depicted in Figure 2.

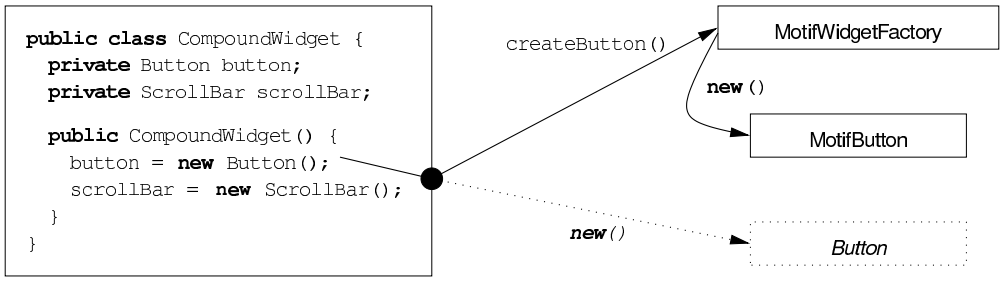
These three boundary maps are grouped into a construct called a *mapset*, as shown in Figure 3. The mapset below is named `widgetFactory` to allude to its purpose. Mapsets are specified in files separate from source code files.

The first boundary map translates parameterized calls into the `CompoundWidget` constructor. Since no parameterized constructor exists within the boundary, such calls must be replaced by calls to the non-parameterized constructor. The communications to be captured by this boundary map are specified in the *capture clause* to the right of the colon on the line below; this boundary map intercepts in-bound messages to `CompoundWidget` parameterized by `WidgetFactory` so the capture clause uses the keyword **`in`**. Each boundary map must declare its result type, even when we are dealing with a constructor; the result type here is `CompoundWidget`, declared at the beginning of the line below. Once a communication conforming to the description in the capture clause crosses the boundary, it is captured and the *body* of the boundary map (declared within a matching pair of braces) is executed in its place. The body of this boundary map simply indicates that the non-parameterized constructor should be run in place of the original call. The argument of type `WidgetFactory` passed in the original call is discarded.

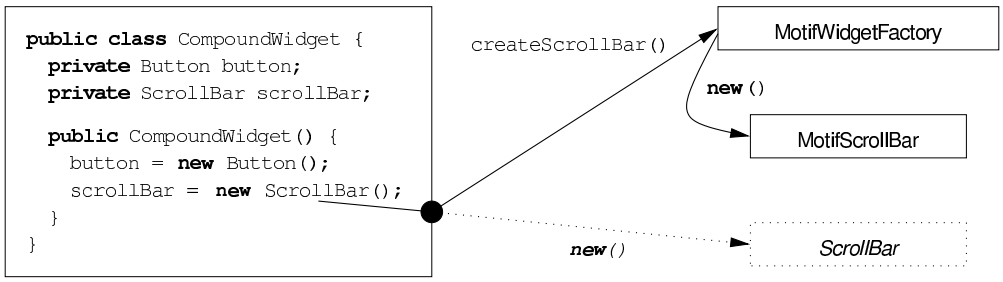
The second boundary map specifies the translation of requests to instantiate `Button`. Any communications passing out across the boundary that are to instantiate `Button` are captured by this boundary map, so it uses the keyword **`out`** (additional forms are available for capturing each access that **`gets`** or **`sets`** the value of a field). An object of type `Button` will be returned to the site that sent the captured communication. We need to replace the call to instantiate `Button` with the appropriate call on an instance of `WidgetFactory`, but no such instance is lying about for us to use. Therefore, we query communica-



(a)



(b)



(c)

Figure 2: The operation of boundary maps attached to `CompoundWidget`. In (a), messages from the external system that attempt to instantiate `CompoundWidget` via a call to a parameterized constructor (which does not actually exist) are rerouted to the non-parameterized constructor. In (b) and (c), attempts to instantiate `Button` and `ScrollBar` within the boundary are rerouted via an instance of a widget factory. (Note that the use of communication history, described in the main text, is not depicted here.)

```
mapset widgetFactory {
  // First boundary map: eliminate passed argument
  CompoundWidget map(): in(CompoundWidget(WidgetFactory)) {
    // Call constructor, discarding the passed argument
    this();
  }

  // Second boundary map: capture attempts to instantiate button
  Button map(): out(Button.new()) {
    // Retrieve the argument passed earlier but discarded
    WidgetFactory fact = (WidgetFactory)
      History.lastInstancePassed(WidgetFactory.class);
    // If no argument was found, create default factory
    if(fact == null)
      fact = new MotifWidgetFactory();
    // Use the factory to create button
    return fact.createButton();
  }

  // Third boundary map
  ScrollBar map(): out(ScrollBar.new()) {
    // Again retrieve the argument passed earlier
    WidgetFactory fact = (WidgetFactory)
      History.lastInstancePassed(WidgetFactory.class);

    // If no argument was found, again create default factory
    if(fact == null)
      fact = new MotifWidgetFactory();

    // Use the factory to create scroll bar
    return fact.createScrollBar();
  }
}
```

Figure 3: The mapset for reconciling inconsistencies in the example. See main text for explanation.

tion history to retrieve an instance of `WidgetFactory`. Specifically, we find the instance that has most recently been passed in any communication.[1] Communication history queries are implemented through a set of methods on the `History` class, provided by IConJ 0.1. This particular query requires an instance of type `Class` and returns an instance of type `Object`; since we know that this must actually be an instance of type `WidgetFactory`, we downcast the returned object. The first call to construct an instance of `CompoundWidget` may come before the first call to construct an instance of `WidgetFactory`; the communication history query would then return **`null`**. In such a case, we choose to create a default widget factory of the Motif variety.[2] Finally, we use the retrieved (or created) widget factory to create and return a button object that is platform-specific.

The third and final boundary map is much like the second, the only differences being that the communications captured are to instantiate `ScrollBar` and that the call to the widget factory creates and returns an instance of `ScrollBar`.

Boundary maps can sometimes be specified generically in order to reuse them in conjunction with multiple maps. Therefore, we must indicate to the tool which mapsets should be attached to which boundaries; this is done with tool directives. The following tool directive indicates that the mapset named `widgetFactory` should be attached to the boundary named `CompoundWidget`. Tool directives are specified in one or more files separate from those containing source code or boundary maps.

```
apply widgetFactory to CompoundWidget;
```

To apply implicit context in this situation, IConJ 0.1 is run with the names of the three files containing the tool directive, the `widgetFactory` mapset, and the `CompoundWidget` source code. The tool produces a new set of Java source code that is ready for compilation. This new source code is not intended to be human readable; if changes are needed, the tool is re-run with changed inputs. The resulting source code simulates the effects of the implicit context model: the `CompoundWidget` module is oblivious to the presence of the Abstract Factory design pattern, while the system as a whole sees this design pattern in use.

4 Tool Structure

Figure 4 shows an overview of the operation of IConJ 0.1. The system composer provides three kinds of inputs to the tool: the original Java source code on which to apply implicit context, the boundary maps describing the use of implicit context, and the tool directives indicating which boundary maps attach to which modules within the source code. Each kind of input is provided in one or more files separate from the other kinds of input.

The tool parses the original source code and boundary maps and forms abstract syntax trees (ASTs) of the code found there. The parser is generated by JJTree and JavaCC[3] from an annotated Java grammar. This grammar is a modified form of the one provided in the JavaCC distribution; it has been tweaked to make the resulting ASTs more explicitly represent such things as method invocations.

According to the tool directives given to the tool, the boundary maps are combined with the original source code. To do this, the tool replicates pieces of the boundary map AST and inserts them at particular points of the source code AST; further details of this process are provided in Section 8.1.

The combined AST is then traversed and modified in order to support communication history queries. Instructions are inserted at particular nodes within the AST to record the communication history; further details of instrumentation and communication history support are provided in Sections 5.5 and 8.2.

[1] The selection of this particular instance of `WidgetFactory` is a design choice on the part of the system composer who creates the boundary maps; other choices are possible using other communication history queries, default values, etc.

[2] Again, this is not the only choice available.

[3] Both available from Sun Microsystems.

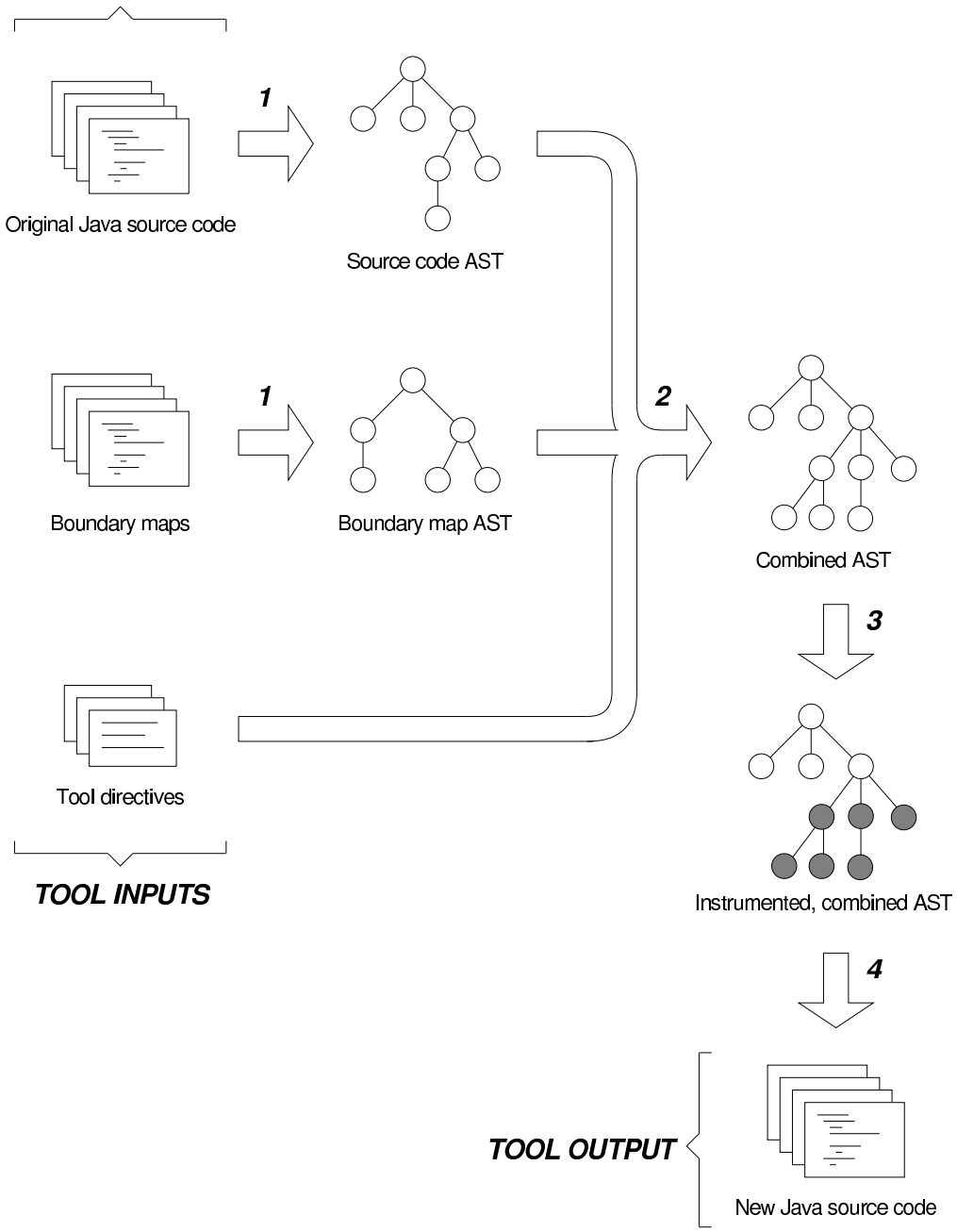


Figure 4: Overview of the operation of the prototype tool. See the main text for an explanation of the four steps involved.

mapset ::=
    mapset *IDENTIFIER* "{"
        { *boundary-map* }
    "}"

boundary-map ::= *standard-boundary-map* | *renaming-map*

standard-boundary-map ::=
    [*modifier*] **map** *FORMAL-PARAMETERS* "**:**"
        capture-clause [*THROWS-CLAUSE*]
    CONSTRUCTOR-BODY

Figure 5: BNF syntax for mapsets. The production rules for the non-terminals *IDENTIFIER*, *FORMAL-PARAMETERS*, *THROWS-CLAUSE*, and *CONSTRUCTOR-BODY* are identical to those for identifiers, formal parameters, method throws, and constructor bodies in Java [Gosling *et al.*, 2000: §3.8, §8.4.1, §8.4.4, & §8.8.5 respectively]. Other non-terminals are defined later in this report.

The final, instrumented AST is then translated back into Java source code. This new version of the source code is not intended for human consumption. If changes are needed to the resulting system (due to error correction or evolution), the inputs to the tool are altered and the tool re-run. In fact, the instrumentation and other transformations make the new source code difficult for a human to read. The tool could as well compile the code as produce transformed source; the source-level transformation approach was chosen to simplify the implementation and debugging of the tool.

5 Boundary Maps

For convenience, rather than naming each individual boundary map, boundary maps are collected into sets called *mapsets*. Each mapset possesses a name, unique among all mapsets, that can be used within tool directives. Below is a simple mapset called `example`:

```
mapset example {
  void map(): in(someMethod()) throws SomeException {
    System.err.println("Calling someMethod()");
    CONTEXT.proceed();
  }
}
```

The `example` mapset contains a single boundary map. Each boundary map consists of up to six parts: a *capture clause*, *formal parameters*, an optional *throws clause*, a *body*, a *result type*, and optional *modifiers*; mapsets conform to the Backus–Naur Form[4] (BNF) syntax [Naur *et al.*, 1960] shown in Figure 5.[5]

The capture clause specifies which communications should be intercepted that cross the boundary to which this boundary map is attached; the `example` specifies that communications should be intercepted that cross the boundary from the external context into the module that are bound for the `someMethod` method. Communication capture is considered in Section 5.1. Boundary maps can allow the arguments of captured communications to be exposed to the body of the boundary map, although the communications captured here possess no arguments; we examine how this is done in Section 5.2.

[4]Surrounding non-terminals with "<" and ">" has been dropped in favour of *slanted* text.
[5]This syntax is intentionally reminiscent of that of AspectJ [Kiczales *et al.*, 2001].

The types of exceptions thrown within the body of the boundary map sometimes need to be explicitly declared within a throws clause, in identical fashion to the rules for Java methods [Gosling *et al.*, 2000: §8.4.4]. The example above declares that the body can throw a `SomeException`.

The body of the boundary map specifies how these captured communications should be altered; in general, each body can consist of an arbitrary block of Java source code, with a few additions and restrictions. In this case, the boundary map has been specified to invoke a print statement prior to the execution of `someMethod`. The body indicates that the original communication should be allowed to proceed on its way through a *call to proceed*, indicated by the statement:

```
CONTEXT.proceed();
```

The call to proceed is used, rather than re-invoking the original method (i.e., calling `someMethod` again). Section 5.3 discusses the reasons for using this construct.

Boundary maps can be prefaced by certain modifiers just as constructor, method, and field declarations can. The permitted modifiers and their interaction with the modifiers of source code declarations are described in Section 5.4. This simple boundary map does not need to make use of communication history, but this is often not the case; we describe how communication history can be used in Section 5.5. An additional form of boundary map is provided by the tool, the *renaming map*; this is described in Section 5.6.

5.1 Capturing Communications

Communications may be captured as they cross boundaries. The capture clause of each boundary map specifies the kind of communications that are captured by that boundary map. While communication capture and alteration are concepts in the dynamic realm, the tool uses and modifies the static properties of the code it operates on to achieve the effect of dynamic capture and alteration. The implementation of this process is described in Section 8.1.

Boundary maps are differentiated on the basis of whether they map communications crossing the boundary from the outside to the inside (in-maps) or vice versa (out-maps). In addition, boundary maps can be defined to select communications that attempt to set the value of a field or to get the value of a field; these are special forms of out-maps. Each of these four options is denoted by a different *event designator*[6] appearing in the capture clause: **`in`**, **`out`**, **`gets`**, or **`sets`**.

Capture clauses conform to the BNF syntax shown in Figure 6. Each event designator is parameterized by a communication description, consisting of the name of the method (or field) being called and a set of *formal arguments*. Each formal argument is either a type name or the name of a formal parameter; this latter kind is described further in Section 5.2.

Not all capture clauses can be used in conjunction with all kinds of boundaries; even of those combinations for which a meaningful interpretation can be defined, not all are supportable in Java. The combinations consist of three dimensions: kind of event designator, communication description properties, and kind of boundary. Tables 1, 2, and 3 describe the treatment of each combination; the four communication kinds are represented as id (bare identifier), $Q.id$ (qualified name), $id(args)$ (unqualified name with formal arguments), and $Q.id(args)$ (qualified name with formal arguments).

For **`in`** event designators, communication descriptions can have prefixes that are identical to the names of the boundaries on which these communications are being intercepted; such prefixes are stripped from the communication description prior to considering how to deal with it. This is where the column marked "—" comes from: if the communication description consists solely of the name of the boundary on which the communication is being intercepted, we have found the final destination for the communication. For example, in-mapping communications of the form `doit()` on the boundary around a method named `doit()`

[6] Strictly speaking, these are not keywords but identifiers treated in a context-sensitive way by the tool's parser; they may be used elsewhere as ordinary identifiers in the boundary map code or source code.

capture-clause ::= *event-designator* "(" *communication-description* ")"

event-designator ::= **`in`** | **`out`** | **`gets`** | **`sets`**

communication-description ::=
    [*qualifier*] (*IDENTIFIER* | **`new`**) [*formal-arguments*]

qualifier ::= *IDENTIFIER* "`.`" { *IDENTIFIER* "`.`" }

formal-arguments ::=
    "(" [*type-or-param-name* { "`,`" *type-or-param-name* }] ")"

type-or-param-name ::= [*qualifier*] *IDENTIFIER* { "`[`" "`]`" }

Figure 6: BNF syntax for capture clauses. Strictly speaking, communication descriptions for **`gets`** and **`sets`** event designators cannot include formal arguments; we avoid this wrinkle for the sake of clarity in this presentation.

	—	id	$Q.id$	$id(args)$	$Q.id(args)$
Class	*?*	*field access*	*field access on nested type*	**method or constructor invocation**	method or constructor invocation on nested type
Interface	*?*	*field access*	*field access on nested type*	*?*	method or constructor invocation on nested type
Method	**method invocation**	*?*	*?*	*?*	*?*
Constructor	constructor invocation	*?*	*?*	*?*	*?*
Field	*field access*	*?*	*?*	*?*	*?*

Table 1: Interpretation and support for capturing communications, for **`in`** event designators, by each kind of boundary. Question marks indicate that there is no simple interpretation for the occurrence of the given communication at the given boundary. **Boldface** entries indicate that capture of the given communication kind is fully supported at the given boundary. *Italicized* entries indicate that capture of the given communication kind is unsupported at the given boundary. Roman entries indicate that capture of the given communication kind is partially supported at the given boundary; see the main text for details.

	id	$Q.id$	$id(args)$	$Q.id(args)$
Class	field access	**field access**	method or constructor invocation	**method or constructor invocation**
Interface	field access	**field access**	method or constructor invocation	**method or constructor invocation**
Method	**field access**	**field access**	**method or constructor invocation**	**method or constructor invocation**
Constructor	**field access**	**field access**	**method or constructor invocation**	**method or constructor invocation**
Field	field access	**field access**	**method or constructor invocation**	**method or constructor invocation**

Table 2: Interpretation and support for capturing communications, for **`out`** event designators, by each kind of boundary. **Boldface** entries indicate that capture of the given communication kind is fully supported at the given boundary. Roman entries indicate that capture of the given communication kind is partially supported at the given boundary; see the main text for details.

	id	$Q.id$
Class	field access	**field access**
Interface	field access	**field access**
Method	**field access**	**field access**
Constructor	**field access**	**field access**
Field	field access	**field access**

Table 3: Interpretation and support for capturing communications, for **`gets`** and **`sets`** event designators, by each kind of boundary. Formal arguments are not permitted in the communication descriptions for **`gets`** and **`sets`** event designators, so no columns are shown for those kinds. **Boldface** entries indicate that capture of the given communication kind is fully supported at the given boundary. Roman entries indicate that capture of the given communication kind is partially supported at the given boundary; see the main text for details.

indicates that the entry in the "—" column should be considered. Similarly, in-mapping communications of the form `SomeClass.doit()` on the boundary around a class named `SomeClass` results in "`SomeClass`" being stripped from the communication; thus, the entry in the column marked "$id(args)$" should be considered. However, if we in-map communications of the form `SomeClass.doit()` on the boundary around a method named `doit()`, no stripping occurs and the entry in the column marked "$Q.id(args)$" should be considered. This stripping process does not have an analogue for out-mapping.

Any communication descriptions matching a table entry that is unsupported results in the tool signalling an error. Communications consisting of just a type name do not occur in Java, so they are unsupported; these correspond to the entries in the "—" column for classes and interfaces. Similarly, the only communications that should cross the boundary of a field, method, or constructor from the outside to the inside are those that name that field, method, or constructor, respectively; all others do not occur in Java. Field accesses cannot be in-mapped in Java, because the Java Virtual Machine does not permit code to be executed when a field is accessed [Lindholm and Yellin, 1999: §3.11.5 & §3.11.8].[7]

The interception of field accesses and method or constructor invocations is partially supported (as indicated in Tables 2 and 3) because we require that such communications actually cross the boundary. For example, a communication that is sent directly by a method to itself is not considered to have crossed the boundary of that method. This is a subtle point: if a method `doit()` contains an invocation `doit()`, it is considered to be sending itself a communication directly. In general, a name reference is considered to cross a boundary if it does not resolve to a declaration within that boundary. The name resolution process is described below. The remaining forms of capturing communications are partially supported due to the way in which communication capture and alteration is implemented; this is discussed in Section 8.

5.1.1 Name Resolution

Declarations are referenced through names; however, a given name can refer to different declarations depending upon the scope in which the name occurs. Therefore, we need a set of rules for determining how to resolve and match names. The following process is used for resolving the names in boundary maps and source code.

Names are resolved with respect to a given boundary; only the declarations inside the boundary are used to resolve the names there. The following code is an example of a complete compilation unit (found within the file `SomeClass.java`) that we will use to illustrate name resolution.

```
package somepackage;

import java.util.Vector;

public class SomeClass extends AnotherClass {
  private Vector set = new java.util.Vector();

  public SomeClass() {}

  public Object doSomething(SomeClass other,
                            SomeClass set) {
    if(other.set.elementAt(0) == this.set.elementAt(0))
      return set.elementAt(0);
    return null;
  }
```

[7] In-mapping of a field can be simulated by out-mapping all accesses to that field—a potentially difficult prospect, since it requires that all source code making such accesses be available to the tool.

```
  private Object elementAt(int index) {
    return set.elementAt(index);
  }
}
```

There are three sites in this source code that reference the `set` field, but each is qualified differently: one is qualified by the local variable `other`, one by the keyword **`this`**, and one is unqualified. Consider the following capture clauses:

1. **`out`**`(set)`

2. **`out`**`(SomeClass.set)`

The expressions that the tool determines as matching each of these capture clauses varies.

If a boundary map containing either of these capture clauses is attached to the boundary of the entire class, none of the three sites in the source code is considered to match. Each communication description resolves to a reference to the field declaration that occurs within the boundary; therefore, these communications would not cross the class boundary.

In the case of boundary maps attached to the boundary around the `doSomething` method, the sites captured by each of these capture clauses is different. The expression `other.set` resolves to `SomeClass.set` since `other` is declared as a formal parameter of type `SomeClass` within the method boundary. The expression **`this`**`.set` resolves to `set`, since **`this`** is no longer needed to disambiguate the field reference from the formal parameter reference. The expression `set` resolves to `SomeClass` since this is a formal parameter reference. Therefore, capture clause #1 would match only the second expression in `doSomething`, **`this`**`.set`, while capture clause #2 would match only the first expression, `other.set`.

Similarly, capture clauses of the forms:

3. **`out`**`(java.lang.Vector.elementAt(int))`

4. **`out`**`(Vector.elementAt(int))`

would capture different expressions if contained in a boundary map attached to the class boundary. Capture clause #3 would match no expressions within this class; it is not until we encounter the import statement that we know that `Vector` resolves to `java.lang.Vector` and this happens outside the class boundary. Capture clause #4 matches three expressions, though: the two within the equality test of the **`if`**-statement, and the one within the body of the `elementAt` method.

5.2 Parameterized Boundary Maps

The body of a boundary map often needs to make use of the objects passed within the communications captured by it. To enable this, boundary maps can be specified with formal parameters; these formal parameters are then bound to arguments within the communication descriptions of the boundary map's capture clause. For example, for communications sent out of a module to the `Vector.elementAt(`**`int`**`)` method and captured, one might be interested in making use of the instance of `Vector` upon which the method is called, the value of the **`int`** argument passed in this call, or both or neither of these. All four of these options can be expressed by boundary maps:

```
Object map(Vector v): out(v.elementAt(int)) { ...

Object map(int pos): out(Vector.elementAt(pos)) { ...
```

```
Object map(Vector v, int pos): out(v.elementAt(pos)) { ...

Object map(): out(Vector.elementAt(int)) { ...
```

The body of the boundary map can then make use of these formal parameters identically to how a method uses its formal parameters [Gosling *et al.*, 2000: §8.4.1].

The order of the formal parameters does not matter. Thus, the third example above could equivalently be:

```
Object map(int pos, Vector v): out(v.elementAt(pos)) { ...
```

The tool currently requires that each formal parameter to a boundary map have its name occur exactly once within the capture clause.

Whether the objects passed in a communication are exposed to the body of the boundary map or not, the set of communications deemed to match the capture clause is unchanged. The tool attempts to resolve those names occurring within the capture clause to declarations within the scope of the boundary map declaration. After this attempt, it then looks for matches in the source code.

5.3 Resuming Captured Communications

Boundary maps capture particular communications. Often, we wish to add functionality to the execution of the methods receiving these communications rather than replacing the original functionality outright. However, directly re-calling the method that has been captured causes a recursive call that is itself captured by the boundary map. Thus, we need a means to avoid the resulting infinite loop. This means is the *call to proceed*, indicated by the expression:

```
CONTEXT.proceed(args)
```

The *args* are a comma-separated list of arguments of identical type and order as the formal parameters of the boundary map. The result type of the call to proceed is that of the captured communication had it not been captured. BNF syntax for calls to proceed is given in Figure 7.

For example, consider the following boundary map:

```
Object map(): out(Vector.elementAt(int)) {
  LockManager.getReadLock();
  // Allow vector access to proceed
  Object obj = CONTEXT.proceed();
  LockManager.releaseReadLock();
  return obj;
}
```

This boundary map captures all attempts to read elements from `Vector`s, ensuring that a read lock is in place before the access happens and that the read lock is released after the access happens.

The functionality of this particular boundary map is not concerned with the actual `Vector` instance on which the access is happening nor with the value of the **`int`** being passed to this method. To do this in the absence of a call to proceed would require a different boundary map, one that exposed these values to the body:

```
Object map(Vector v, int pos): out(v.elementAt(pos)) {
  LockManager.getReadLock();
  // Allow vector access to proceed
  Object obj = v.elementAt(pos);
  LockManager.releaseReadLock();
  return obj;
}
```

PRIMARY ::=
    call-to-proceed
    | *PRIMARY-NO-NEW-ARRAY*
    | *ARRAY-CREATION-EXPRESSION*

call-to-proceed ::=
    CONTEXT "**.**" `proceed` "`(`" [*ARGUMENT-LIST*] "`)`"

Figure 7: BNF syntax for calls to proceed. The Java syntax for primary expressions (indicated by the non-terminal *PRIMARY*) [Gosling *et al.*, 2000: §15.12] has been extended to include calls to proceed. The other non-terminals in *slanted* text correspond to existing Java syntax.

But this makes the code more complicated, runs the risk that we may inadvertently alter the values of `v` and `pos`, and causes an infinite loop because the call to `Vector.elementAt` within the boundary map body is captured by the boundary map as well.

If we wished to replace the `Vector` object and **`int`** value passed in the call to `elementAt`, we could use the following boundary map.

```
Object map(Vector v, int pos): out(v.elementAt(pos)) {
  LockManager.getReadLock();
  Vector special = getSpecialVector();
  Object obj = CONTEXT.proceed(special, pos + 1);
  LockManager.releaseReadLock();
  return obj;
}
```

In this example, we have changed the access attempt to have it made on a different vector than the one originally called, and the position at which the access is made is one greater than the original. The call to proceed must be provided with arguments of the same types and same order as occur in the formal parameters of the boundary map.

The use of the call to proceed places an onus on the system. Boundary maps can be used to effectively introduce methods to a class that do not already exist. Should such a boundary map make a call to proceed, there is no pre-existing implementation that should be executed. There are two possible options in such a situation: ignore the call to proceed, or signal an error. The tool does the latter.

5.4 Modifiers

Boundary maps translate communications between differing world views. Constructor, method, and field declarations can all possess modifiers. Therefore, to be a true translation, boundary maps need to cope with any difference in which modifiers are perceived to affect behaviour. The boundary map should declare itself to possess the set of modifiers expected by the calling context, and should translate these into the set of modifiers provided by the called context. For example, say that the method `doit` is declared only with the modifier **`private`**. If the world view of the calling context expects that `doit` is **`public`**, **`static`**, and **`synchronized`** then the boundary map should be declared **`public`**, **`static`**, and **`synchronized`** while being implemented to translate this view to the reality that `doit` is only **`private`**.

Currently, the prototype tool only supports the **`static`** modifier on boundary maps (see Figure 8 for BNF syntax); issues arising from their use are described in the subsection below. However, some visibility modifiers can be simulated with the tool; these are described in the subsection after next.

modifier ::= **`static`**

Figure 8: BNF syntax for modifiers.

5.4.1 The `static` Modifier

Methods and fields in Java can be declared as operating either in the context of a particular instance, or without such a context. The former case is the default; the latter is indicated by the presence of the **`static`** modifier on the declaration of the method or field. Likewise, boundary maps can be declared as **`static`**.

Specifying a boundary map without a **`static`** modifier implies two things. First, the communication being captured must possess an instance context; for an out-map, this is the instance context from which the communication was sent. Second, this captured instance context can be used within the body of the boundary map via the keyword **`this`** just as is the case with method bodies. Specifying a boundary map with a **`static`** modifier indicates that no instance context is required of the communications captured by the boundary map. However, the keyword **`this`** cannot be used within the body of such a boundary map; its presence will cause a compile-time error.

A non-**`static`** boundary map cannot capture communications that do not provide an instance context. A **`static`** boundary map can capture communications that provide an instance context; this instance context will be discarded. This discarding procedure is consistent with the operation of Java itself [Gosling *et al.*, 2000: §15.12.1].

5.4.2 Simulating Visibility Modifiers

The prototype tool ignores issues of the standard visibility modifiers. For example, it does not take into account the fact that calls to **`private`** members cannot cross class boundaries, and any in-mapped method is considered **`public`**. However, visibility modifiers can be mimicked to a degree.

Consider the following simple class:

```
public class Simple {
  private void doit() {}
}
```

We may in-map calls to the `doit` method, with or without adding behaviour there. This effectively causes `doit` to appear to be **`public`** from the perspective of the external context. So, for example, we might specify a boundary map to be attached to `Simple` as:

```
  void map(): in(doit()) {
    CONTEXT.proceed();
  }
```

The external context would then see the declaration of `Simple` as:

```
public class Simple {
  public void doit() {}
}
```

We can cause methods to be invisible from the perspective of the external context, too. If we wished to make the method `doit` visible only within the boundary, we could in-map calls to it in the following way:

```
  void map(): in(doit()) {
    throw new IllegalAccessError();
  }
```

Instances of the class `IllegalAccessError` are thrown by the Java Virtual Machine when an attempt is made to invoke a method from a context that is not permitted access to it; this implementation mimics that behaviour. This approach is not totally satisfactory, as modules containing calls to `doit` will still compile; only at run-time will they fail.

Out-mapping can similarly be used to alter effective interfaces. Imagine that we have a class `Client` that calls `doit`:

```
public class Client {
  public void someMethod() {
    Simple simple = new Simple();
    simple.doit();
  }
}
```

We can out-map this call to mimic the effect of a **`public`** modifier on `doit`:

```
  void map(): out(Simple.doit()) {
  }
```

If we wished to have this out-map make a call to proceed, though, problems would occur because of the way boundary maps are implemented by the tool. Boundary maps are implemented within the scope of the boundary to which they apply. The in-map is implemented within the scope of the `Simple` class and, thus, is able to access the `doit` method even though this is **`private`**. The out-map is implemented within the scope of the `Client` class and, thus, `doit` remains **`private`** from its perspective. This behaviour is not a deliberate design.

5.5 Communication History

Communication history is, conceptually, a complete record of every piece of information passed between modules. The prototype tool deals strictly with method and constructor invocations, though. These are recorded in two fashions: on a causal basis, in terms of a tree-structure consisting of calls and returns for each thread of control within an execution; and on a temporal basis, irrespective of causality. A means of querying communication history on either basis is supported by the prototype tool, through an interface on a specially provided class called `History` and its attendant helper, `Call`. For each invocation, the parameters passed can be retrieved through the appropriate queries. This is meant literally: in Java, objects are passed by reference; therefore, the *reference* that has been passed can be retrieved, not necessarily the original state behind that reference.

There are several methods defined on `History` for querying communication history, as listed in Table 4; these are all static methods. Most of these allow the thread, class, method name, and method formal parameter types to be optionally specified; **`null`** arguments are treated as though any value in the corresponding position matches. All these methods return an instance of `Call` if a valid match is found, or **`null`** otherwise. A sample method, `lastCall`, is shown and described in Figure 9. The methods defined on the `Call` class are shown in Table 5. The complete interface to the `History` and `Call` classes is described in Appendix C.

To examine the operation of communication history and its query interface, consider the following toy program.

```
public class Test {
  public static void main(String[] args) {
    Test test = new Test();
    test.printIt(1);
```

Method	Search kind
`firstCall`	either
`lastCall`	either
`lastCallAnySubclass`	either
`lastCallInCFlow`	causal
`lastCallReturningClass`	either
`lastInstancePassed`	temporal
`mostRecentCall`	temporal

Table 4: The query methods defined on the `History` class. Each method searches causally or temporally. Those methods marked as "either" have forms that can operate in either of these fashions. The interface is described in detail in Appendix C.

Result type	Method name
`java.lang.String`	`getMethodName()`
`java.lang.Thread`	`getThread()`
`java.lang.Object`	`getReturnValue()`
`java.lang.Object`	`getTarget()`
`java.lang.Class`	`getTargetClass()`
`java.lang.Object`	`getParameter(int i)`
`Call`	`getPredecessor()`
`Call`	`getParent()`
`boolean`	`precedes(Call other)`

Table 5: The interface to the `Call` class. The interface is described in detail in Appendix C.

lastCall

```
static Call lastCall(Thread thread,
                     Class cls,
                     String mth,
                     String[] paramTypes,
                     Object returnVal,
                     Call relativeTo)
```

Searches for and returns the most recently occurring call conforming to the search parameters passed as arguments; returns **null** if no call can be found that matches these parameters.

Parameter	*Description*	*Default*
`thread`	The thread of control whose tree should be searched, in the case of causal searches; or **null** if temporal searches should be performed	current thread
`cls`	The exact class on which the executed method was defined; or **null** if the class is not of interest	—
`mth`	The name of the method on which the executed method was defined; or **null** if the method is not of interest	—
`paramTypes`	An array containing the names of the types of the formal parameters as they appear lexically in the method declaration; or **null** if the parameter types are not important	—
`returnVal`	The object that must have been returned by the method execution; or **null** if the returned object is not of interest	**null**
`relativeTo`	The call at which the search should begin; the predecessors of this call are compared, in temporal order, to the search parameters	`mostRecentCall()`

Figure 9: A sample query method on the `History` class. The *slanted* formal parameters are optional; that is, the `lastCall` method is overloaded with forms missing these parameters, in all combinations. If an argument is not provided that corresponds to an optional formal parameter, the default value indicated in the table is used.

```
    test.printIt(2);
    aMethod();
    test.printIt(3);
  }

  public void printIt(int number) {
    System.out.println(number);
  }

  public static int aMethod() {
    Test otherTest = new Test();
    otherTest.printIt(2);
    return 0;
  }
}
```

Executing this program causes `Test.main` to be invoked initially. A series of numbers is printed to the standard output in the order: 1, 2, 2, 3. The communication history just before the program terminates is shown in Figure 10.

We can see the effects of communication history by applying the following mapset to the boundary of `main`:

```
mapset findIt {
  static void map(): in(main(String[])) {
    CONTEXT.proceed();

    String[] params = new String[] { "int" };
    Call call =
      History.lastCall(Test.class, "printIt", params);
    call =
      History.lastCall(Test.class, "printIt", params,
                       call.getPredecessor());
    call = call.getParent();

    System.out.println(call.getMethodName() + ": " +
                       call.getReturnValue());
  }
}

apply findIt to Test.main(String[]);
```

This boundary map begins by calling to proceed with the original functionality of `main`. It then retrieves the most recent call to `printIt`, which can be seen from Figure 10 to be the lowermost call on the diagram. Tracing back over the sequence of dashed arrows, the boundary map finds the call that occurs most recently prior to this one, which is the rightmost call on the diagram. The parent of this second most recent call to `printIt` is retrieved by tracing back over the solid arrow. Finally, the boundary map prints the name of this parent and the result it returns, as:

```
aMethod: 0
```

At the time of developing the boundary map, we may know that the call we are interested in retrieving is the last call to `aMethod`. In this case, a simpler query to communication history could be used in the boundary map:

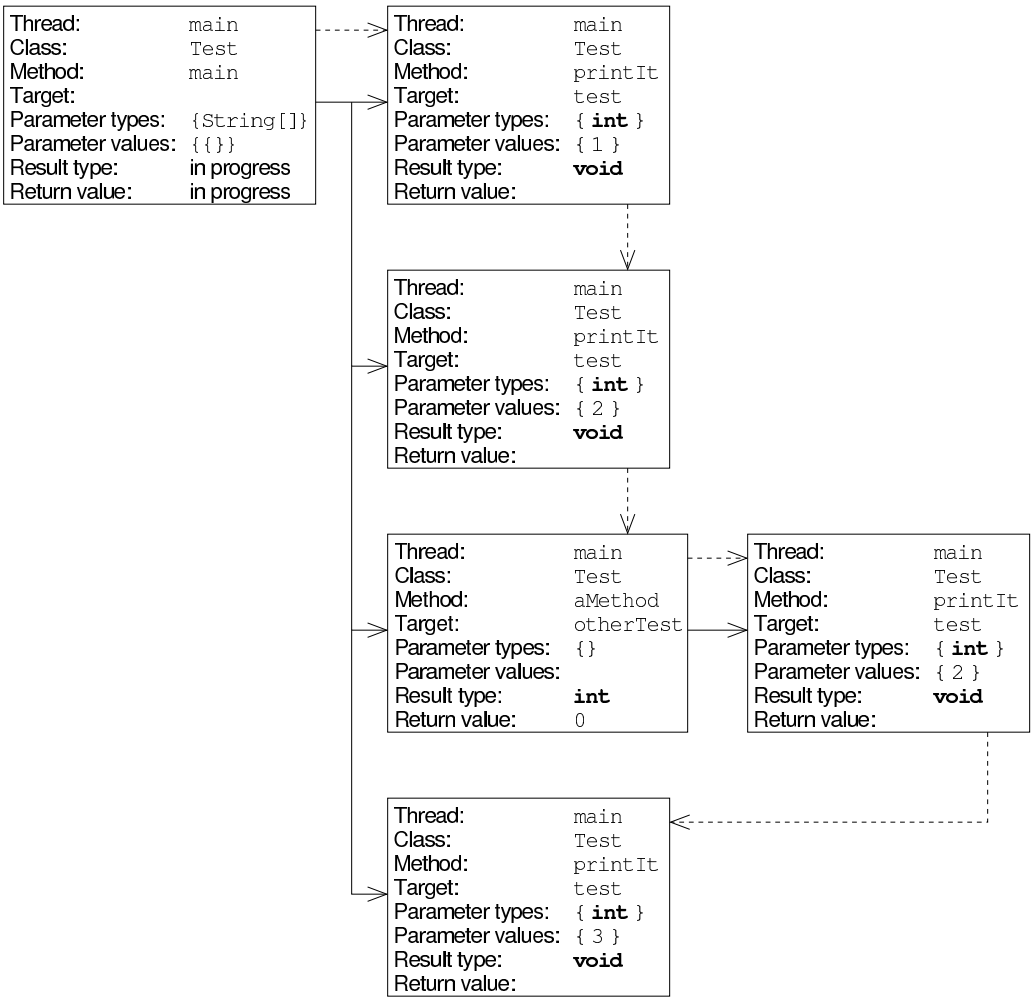


Figure 10: An example communication history. The solid arrows indicate the causal relationships between the calls; these arrows lead from the callers to the callees with later callees lower on the diagram. The dashed arrows indicate the temporal ordering of the calls, leading from earlier to later calls. Since the execution of `main` has not finished, its result is not recorded. Calls to constructors have been left off the diagram. The target objects are actually stored as references; the names `test` and `otherTest` are shown here for readability.

```
mapset findIt2 {
  static void map(): in(main(String[])) {
    CONTEXT.proceed();

    String[] params = new String[0];
    Call call =
      History.lastCall(Test.class, "aMethod", params);

    System.out.println("aMethod: " +
                       call.getReturnValue());
  }
}
```

This locates the same, ultimate call of interest and produces the same output as the earlier version.

The communication history functionality was provided in this class-based fashion for the sake of ease of development of the proof-of-concept tool. It being initially unclear how communication history was to operate and be maintained, we decided that a class-based approach would allow maximum flexibility with minimum changes needed to the tool as the research progressed. The interface that has been implemented for communication history is, by no means, complete. For example, there are currently no query methods that allow one to specify that a call should be found that contains a particular object as an argument, and there is currently no method for determining the number of parameters on a returned call. Section 8.2 discusses the drawbacks of the current implementation approach; Walker and Viggers [2004] describes a means for eliminating these difficulties.

5.6 Renaming Maps

The boundary maps described thus far are based upon the notion that communications are captured as they cross boundaries, and that an operational description of the alteration of these communications can be specified. The pragmatics of Java make the use of this model difficult for altering communications involving type references, such as static method invocations (e.g., capturing the reference to `System` within an invocation to `System.loadLibrary`). Type references are not first class entities in Java, so replacing one type reference with another is not possible procedurally. Although Java provides support for introspection, one cannot easily transform a static type reference embedded within a potentially complex expression into a different expression that uses introspection. Instead, the prototype tool provides a second kind of boundary map: renaming maps.

A renaming map specifies a target name to be matched and a replacement name that is inserted in its stead. For example, if we wished to replace all references to `java.lang.System`, within a boundary, with a reference to `SomeClass`, we would use the following renaming map:

```
java.lang.System -> SomeClass;
```

This would replace any *explicit* occurrences of the string "`java.lang.System`," and hence, implicit references through the string "`System`" will not match. That is, name resolution does not occur.

In general, the tool allows any type name to be replaced with any other type name (although array types are not supported). However, Java syntax is ambiguous: type names and field accesses have identical syntax. If a name does not resolve at a given boundary, we often cannot tell whether it refers to a type or a field. Therefore, the tool considers any unresolved name to be a fair target of renaming maps. BNF syntax for renaming maps is provided in Figure 11.

Renaming maps do not, by themselves, ensure the type compatibility of all expressions upon which they act. A local variable whose type is altered by a renaming map may have assigned to it an expression

renaming-map ::= *possible-type-name* "->" *possible-type-name* ";"

possible-type-name ::= [*qualifier*] *IDENTIFIER*

Figure 11: BNF syntax for renaming maps.

tool-directive ::= **apply** *IDENTIFIER* **to** [*qualifier*] *IDENTIFIER* ";"

Figure 12: BNF syntax for tool directives.

whose type is not assignable to the replacement type. For example, consider the following source method and renaming map:

```
public void someMethod() {
  String s = "test string";
}

String -> mypackage.String;
```

The body of `someMethod`, after applying the renaming map, effectively becomes:

```
  mypackage.String s = "test string";
```

However, this statement is illegal because an instance of `java.lang.String` is not assignable to a variable of type `mypackage.String` since the former is not a subtype of the latter. If the tool allowed assignment operations to be captured and altered (which it does not), the assignment could presumably be replaced; otherwise, the resulting module will not compile. This behaviour is peculiar to `java.lang.String` since it is the only class for which literals are defined within Java.

More generally though, one would need to assure that the types of both sides of assignments were compatible. The tool does not currently flag such constraint violations, but allows the compiler (used after the tool has produced transformed source) to catch them. The presence of such incompatibilities means that the translation between world views internal and external to the module is incomplete.

6 Tool Directives

Tool directives are statements telling the tool which mapsets to attach to which boundaries, like the following:

```
apply someMapSet to SomeClass;
```

Each tool directive consists of two parts: the name of the mapset (`someMapSet`, in this case) to apply to a boundary, and the name of the boundary itself (`SomeClass`). BNF syntax for tool directives appears in Figure 12.

The prototype tool allows three kinds of boundaries to be named within tool directives: whole classes, individual methods and constructors, and individual fields. This is not to say that implicit context is fully supported at all these boundaries; see Section 5.1 for more detail. If the tool is unable to locate the named boundary within the files provided to it, it prints a warning to this effect.

Tool directives reside in the global namespace. This means that mapsets and boundaries must be named within them through global names, i.e., they must be fully qualified. For example, the class `JavaOutlinePage` of Eclipse occurs in the package `org.eclipse.jdt.internal.ui.javaeditor`; thus, it must be referred to as `org.eclipse.jdt.internal.ui.javaeditor.JavaOutlinePage` within a tool directive.

Fields are referenced by appending their names to the fully qualified name of their defining class.

Methods also are referenced by appending their names to the fully qualified name of their defining class, but are more problematic because of the formal parameter list of each. To match a method, the tool lexically matches the types of the formal parameters as specified within the source and within the tool directives; no resolution of the types found in either formal parameter list is performed prior to matching. For example, consider this simple example of Java source:

```
package somepackage;

import otherpackage.SomeClass;

public class AnotherClass {
  public AnotherClass(SomeClass arg) {}
  public void doit(SomeClass arg) {}
}
```

and this tool directive:

```
apply someMapSet to somepackage.AnotherClass.
                      doit(otherpackage.SomeClass);
```

Although the tool would search within `AnotherClass` here to find the `doit` method, it would not find any matches since the lexical declaration of `doit` says that it takes a formal parameter of type `SomeClass` and not `otherpackage.SomeClass`—even though the **`import`** statement says that this is the class to which `SomeClass` resolves. The tool only uses the information provided within the files passed to it, but it does not assume that the information contained there forms a closed universe. Therefore, we chose the parameter type matching rule to avoid the need for disambiguating so-called on-demand import statements, such as

```
import otherpackage.*;
```

In the presence of this alternative statement, `SomeClass` could resolve to `SomeClass`, `otherpackage.SomeClass`, `somepackage.SomeClass`, or even `java.lang.SomeClass`.

Constructors are handled similarly to methods. The name of a constructor is the same as the unqualified name of its class. For example, the constructor shown above for `AnotherClass` would be referenced in a tool directive as:

```
somepackage.AnotherClass.AnotherClass(SomeClass)
```

Since tool directives are separated from the modules they affect, we chose to place tool directives in the global namespace; placing them inside particular source modules seems inappropriate. This choice has additional consequences: not all boundaries are easily named from the global namespace. For example, Java possesses constructs such as local classes whose names are not well-defined outside their local scopes, and anonymous classes which have no names at all. While we can define an arbitrary rule to construct their names (such as a numbering scheme based on their lexical order within a globally named scope), these resulting names tend to be unstable: if we merely reorder the source code, the names will change. Similar problems would arise should we choose to support sub-method-level boundaries, such as boundaries around individual statements. It is possible that specifying boundaries through operational means—defining properties or conditions that need to be met—would allow such problems to be overcome. However, it is not clear at this point whether separate boundaries around such difficult-to-name modules are needed.

7 Sequencing Boundary Maps

More than one boundary map can readily be defined that attempts to capture a given communication. In these situations, it is important to have an explicit sequence in which boundary maps are to apply. A total order of boundary maps is defined for the prototype tool in the following fashion: of greatest precedence is the order in which the files containing tool directives are specified to the tool, followed by the order of the tool directives within a particular file, followed by the lexical order of boundary maps within a mapset named by a tool directive.

The tool blindly applies the boundary map of greatest precedence to the source before considering where or how the second most precedent boundary map is to apply. The source that is input to the tool for matching against the second boundary map is the output of the first.

For example, consider the following source, mapset, and tool directive.

```
public class SomeClass {
  public void method1() {
    System.err.println("In method1()");
  }

  private void method2() {
    String s = "test string";
  }
}

mapset example {
  void map(): in(method1()) {
    System.err.println("Before method1()");
    CONTEXT.proceed();
  }

  void map(): in(method1()) {
    System.err.println("Ignoring method1()");
  }

  String -> mypackage.String;

  String -> java.lang.String;
}

apply example to SomeClass;
```

The tool would apply the first in-map on `method1` first, followed by the second in-map on `method1`. As a result, calls to `method1` would result in the string "`Ignoring method1()`" being printed. Next would come the first renaming map, which would result in the type of the local variable `s` becoming `mypackage.String`. Finally, the second renaming map would be applied, but would not find any matches for "`String`" since the first renaming map has already replaced these.

8 Implementation Issues

The previous sections within this report have largely described the features of the prototype tool from the perspective of a system developer using it. However, this leaves the question of how the tool actually performs its tasks, a question that leads to subtle properties of behaviour to which we have alluded.

Recall that every boundary map possesses a body that can contain fairly arbitrary Java source code. Once a boundary map has captured a communication, its body indicates how that communication is to be altered. However, this is a model that must operate in the context of the realities of Java, where one cannot literally intercept communications (especially to non-existent methods) as they cross boundaries, and where it would be very inefficient to actually do so. Instead, as indicated in Section 4, the tool performs a multi-stage process before compile-time on the three kinds of inputs provided it.

We begin, in Section 8.1, by examining how the tool combines boundary maps and Java source to simulate the interception of communications crossing boundaries. Section 8.2 looks at instrumentation of the resulting source to support communication history, and the effects that sequencing multiple boundary maps has upon this support.

8.1 Combining Boundary Maps and Java Source

The tool combines specified boundary maps with the source within specified boundaries. The way it goes about this depends on two things: the kind of boundary and the kind of boundary map to be combined. We begin by looking at the combination process for in-maps, followed by the combination process for out-maps (including **`gets`** and **`sets`** event designators). Table 1 indicates the support provided by the tool for various combinations of kinds of communication descriptions and boundaries.

In-mapping method invocations results in two possible situations: either the method is already present, or it is not. If it is already present, we change the name of the method to hide it[8] and any references to that method within the boundary are changed to the new name of the method. In both cases, the body of the boundary map is inserted as the body of a new method, with the original name of the replaced method. The result type and context-kind of this new method are taken from the result type and context-kind of the boundary map; after all, the purpose of the boundary map may be to alter these properties to the perspective of a different world view. The new method is inserted at the level of the closest enclosing class, even when the boundary map is applied to a method boundary.[9]

In-mapping constructors is more complicated, because we are stuck with the name of the constructor. Instead, we can play with the number and types of parameters that the constructor takes; dummy parameters can be inserted to effectively hide the original constructor, with appropriate dummy arguments inserted in calls to it within the boundary. This has not been fully implemented in the tool.

Attempts to capture qualified method or constructor invocations can be problematic. When applied to the boundary of a type, they are interpreted as method or constructor invocations on nested types. This is done as in the non-qualified case as long as the nested types are *already present*. We deemed it too complicated to introduce new classes correctly for it to be worthwhile doing so in the prototype; it is not clear from whence some details about the new class would come.

Out-maps are generally more straightforward. Again, the body of the boundary map is inserted as a new method with an obfuscated name.[10] All call sites (or field access sites) within the boundary matching the communication description are replaced by calls to the new method.

Out-mapping constructors has a few more twists to it. In Java, there are two components to an invocation of a constructor. First, space is allocated to the new instance via use of the **`new`** operator, then this space is initialized through the actual constructor invocation. If we were to only capture the latter component, we would need to constrain our out-map to ensure that it consisted of only a different constructor invocation. Instead, the tool allows for the capture of the **`new`** operator itself, through a special syntax. To

[8] The new name consists of the prefix "`__CONTEXT$inner`" followed by a number, which starts at 0 for the first changed method, 1 for the second, etc.

[9] One cannot embed a method directly inside of another method in Java—it is not clear what the semantics of such an embedding would be.

[10] Here, the name consists of the prefix "`__CONTEXT$map`" followed by a number.

capture constructor invocations of the form **new** `somepackage.SomeClass(`*args*`)`, one describes the communication as `somepackage.SomeClass.new(`*args*`)`. Each entire instantiation expression within the boundary is replaced in this situation.

If the boundary map body contains a call to proceed, this must be replaced with an invocation of the original method (via its new name, in the case of in-maps) or original constructor, or with an access of the original field, where the communication arguments that are exposed to the boundary map body are replaced with whatever new values are in place and the other arguments are unchanged. If no such original target exists, the tool signals an error.

8.2 Instrumentation, Communication History, and Sequencing of Boundary Maps

In order to reflect upon the history of communications made within a system, we need both a means to record the communications made within that system, and a means to access this record. The proof-of-concept implementation of communication history for Java stores method calls and method returns within a data structure consisting of one tree per thread of control, and a doubly-linked list through the nodes in all these trees to indicate the temporal order of the events therein. Interfaces to this data structure are provided for the system developer to utilize communication history; see Section 5.5 for details.

To store calls and call returns in the tree, we defined two snippets of code to instrument the methods in a system, one that is executed at the start of each method and one that is executed at the end of each method. These snippets need to worry about operating correctly in the presence of exceptions, and thus, tend to be long and ugly looking; Appendix B shows a simple example in all its gory detail.

Every method in the classes operated on by the tool is instrumented. This means that every method invocation causes information to be stored in the communication history record, and it is stored for the entire duration of the execution of the system. This is obviously inefficient both in terms of execution speed and memory requirements.

More subtle problems arise from the fact that instrumentation does not occur until after all boundary map transformations have occurred, and that only method entries and exits are instrumented, not the call sites themselves. Consider the following source, mapset, and tool directive.

```
public class SomeClass {
  public static void main(String[] args) {
    OtherClass obj = new OtherClass();
    for(int i = 0; i < 3; i++)
      obj.method1();
  }
}


public class OtherClass {
  public void method1() {
    method2();
  }

  public void method2() {
    String s = "a string";
    System.err.println(s.length());
  }
}
```

```
mapset example {
  void map(): in(method1()) {
    Call call = History.lastCall(null, "method2", null);
    if(call == null)
      CONTEXT.proceed();
    else
      System.err.println("method2() called before");

    Call call2 =
      History.lastCall(String.class, "length", null);
    System.err.println(call2 != null);
  }

  void map(): in(method2()) {
    CONTEXT.proceed();
  }
}


apply example to OtherClass;
```

Implicit context is being used here (through the first boundary map) to change the behaviour of the system such that `method2` is invoked at most once. The second boundary map is present simply to cause the problem illustrated in the next paragraph; conceptually, its effect should be idempotent.

Before implicit context was applied, running `SomeClass` would have caused the following to be printed:

```
8
8
8
```

We would expect that, after applying implicit context, this would become:

```
8
false
method2() called before
true
method2() called before
true
```

However, this will not happen under the prototype tool. Instead, this will be printed:

```
8
false
8
false
8
false
```

Appendix B shows the code that results from the application of the tool on this example. There are two problems, as described below.

The in-map on `method2` causes it to be renamed to `__CONTEXT$inner1` and the call site within `method1` is changed to this new name. A new method is inserted in `OtherClass` called `method2`

that delegates to `__CONTEXT$inner1`, replacing the call to proceed within the boundary map. But when the resulting class is instrumented, the method that is getting invoked each time is not called `method2` but `__CONTEXT$inner1`—we have changed the name out from under our communication history query. This is why "`method2() called before`" is never printed.

The other problem arises from the fact that the call site within `method2`, invoking `String.length()`, is not instrumented and the source for `String` itself has not been fed into the tool. Therefore, the query to find the last call on `String.length()` always fails, returning **`null`**.

All these difficulties arise from the simplistic implementation of communication history that we chose. This sufficed for proof-of-concept investigations, but does not suffice for an industrial strength version of the tool.

9 Conclusion

We have described IConJ 0.1, a proof-of-concept tool for the application of the model of implicit context to Java source code. IConJ 0.1 is used for generative adaptation of Java source code. It may be used to overcome inconsistencies between source fragments that are to be composed into a functioning system.

A system composer must describe four things in specifying and using IConJ 0.1: (1) the Java source code on which implicit context is to operate; (2) to which boundary or boundaries it must be attached; (3) which communications it is to intercept; and (4) how it should alter the intercepted communications. Items 3 and 4 are provided by the boundary map construct; multiple boundary maps are organized into named mapsets. Item 2 is provided by tool directives, which name mapsets and the boundaries to which they are to apply.

IConJ 0.1 performs a multi-stage process before compile-time on the three kinds of inputs provided it (namely, tool directives, boundary maps, and original Java source):

1. The boundary maps and original Java source are parsed and stored as abstract syntax trees (ASTs).
2. Each tool directive is parsed to identify the mapset and boundary named in it. These names are matched against the ASTs.
3. Assuming that the mapset and boundary matching the names within the tool directive are found, each boundary map within the mapset is applied to the boundary, in the lexical order in which they occur. After each transformation, the transformed source AST is used as the input for the next transformation.
4. After all the tool directives have been carried out, the transformed source AST is instrumented to support communication history queries.
5. The transformed source AST is unparsed to produce a new set of Java source, ready for compilation.

Boundary maps consist of: a capture clause, describing the set of communications to be intercepted at the boundary to which the boundary map is attached; a body, specifying how the intercepted communications are to be altered; a set of formal parameters, bound to parts of the intercepted communications and exposed for use within the body; and various other minor parts, such as a result type, and optional **`throws`** clause and/or **`static`** modifier.

Communication history can be used within boundary map bodies. It is supported through a set of methods on two special classes, called `History` and `Call`. Communication history is supported by instrumenting the transformed source code in order to record every method entry and exit. This simplistic approach has several drawbacks, including impact on the speed of execution, and ever-increasing memory

demands the longer the execution continues. Walker and Viggers [2004] describe an improved mechanism that avoids these difficulties.

While IConJ 0.1 sufficed to explore the concepts of implicit context for which it was constructed, it possesses a variety of shortcomings that would prevent its adoption in an industrial setting. Work to construct a stronger version of this system is underway.

References

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, United States: Addison–Wesley.

James Gosling, Bill Joy, Guy Steele, and Gilad Bracha (2000). *The Java Language Specification* (Second edition). The Java Series. Boston, United States: Addison–Wesley.

Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold (2001). An overview of AspectJ. In J. Lindskov Knudsen, editor, *ECOOP 2001—Object-Oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. (15th European Conference on Object-Oriented Programming; Budapest, Hungary; 18–22 June 2001).

Tim Lindholm and Frank Yellin (1999). *The Java Virtual Machine Specification* (Second edition). The Java Series. Reading, United States: Addison Wesley Longman, Inc.

Peter Naur, editor, J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger (1960). Revised report on the algorithmic language ALGOL 60. *Communications of the ACM 3*(5), 299–314.

Robert James Walker (2003). *Essential Software Structure through Implicit Context*. Ph.D. thesis, Department of Computer Science, University of British Columbia, Vancouver, Canada.

Robert J. Walker and Gail C. Murphy (2000). Implicit context: Easing software evolution and reuse. In David S. Rosenblum, editor, *Proceedings of the ACM SIGSOFT Eighth International Symposium on the Foundations of Software Engineering*, pages 69–78. (FSE8; San Diego, United States; 8–10 November 2000).

Robert J. Walker and Kevin Viggers (2004). Implementing protocols via declarative event patterns. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*. To appear.

A Example Inputs

The tool has four inputs, two Java source files (`SomeClass.java` and `OtherClass.java`), one mapset file (`example.map`), and one tool directive file (`apply.app`). The tool makes use of the three letter suffix of each filename in determining the kind of its contents.

A.1 Contents of `SomeClass.java`

`SomeClass` is merely a client of `OtherClass`, instantiating it, and calling its `method1` three times.

```
public class SomeClass {
  public static void main(String[] args) {
    OtherClass obj = new OtherClass();
    for(int i = 0; i < 3; i++)
      obj.method1();
  }
}
```

A.2 Contents of `OtherClass.java`

`OtherClass` possesses two methods: `method2`, which prints the length of a string; and `method1`, which simply delegates to `method2`.

```
public class OtherClass {
  public void method1() {
    method2();
  }

  private void method2() {
    String s = "a string";
    System.err.println(s.length());
  }
}
```

A.3 Contents of `example.map`

The `example` mapset contains two boundary maps. The first of these is responsible for causing `method2` to be invoked at most once. If it finds that `method2` has been called before, it prints a string to this effect. In addition, it retrieves the last call to `String.length`.

```
mapset example {
  void map(): in(method1()) {
    Call call = History.lastCall(null, "method2", null);
    if(call == null)
      CONTEXT.proceed();
    else
      System.err.println("method2() called before");

    Call call2 =
      History.lastCall(String.class, "length", null);
    System.err.println(call2 != null);
```

```
  }

  void map(): in(method2()) {
    CONTEXT.proceed();
  }
}
```

A.4 Contents of `apply.app`

The tool directive causes the `example` mapset to be attached to the boundary of `OtherClass`.

```
apply example to OtherClass;
```

B Example Outputs

When the prototype tool is run on the four inputs in Section A, it produces two new Java source files, one for `SomeClass` and one for `OtherClass`; see Section 4 for an overview of the operation of the tool. So as not to overwrite the original source files, these are stored in a subdirectory of the current working directory called `icworkingdir`.

The code shown in the two subsections below has been cleaned up a bit in two ways for the sake of readability. It has been pretty printed, and the qualifiers for several packages have been elided, namely those for `ca.ubc.cs.se.context.instrument` (the package for the instrumentation classes, including `History` and `Call`) and `java.lang`.

The instrumentation uses a few classes not in the abstract programming interface for call history queries; information that remains the same for each invocation of a given method or constructor is stored in a single object that is reused for each invocation. `CallReceptor` is used for such recurring information, while `CallState` and `ReturnState` are invocation-specific. `CallReceptor` is used to represent the class, method, formal parameter types, and formal parameter names of each method. `CallState` records the object references and primitive values passed to an invocation. `ReturnState` records the result value, be it an actual object reference or primitive value that is returned, or be it an exception that is thrown. Both `CallState` and `ReturnState` record a reference to the target object on which the method is invoked.

`CallReceptor` instances are created and initialized in a **`static`** initializer for each class. An instance of `CallState` and `ReturnState` are created at the entry to each method or constructor.[11] The entry to the method is then recorded in call history.

The general structure of the instrumentation is shown in Figure 13. The original body of the method is placed within the body of a **`try`** statement, so that any exceptions it may throw can be recorded in the `ReturnState` instance for that invocation. The complex set of tests in the **`catch`** clause for each method is necessary because declared exception types must be differentiated from errors and exception types that do not need to be declared. The base class for all exceptions, `Throwable`, is unfortunately an exception type that must be declared; hence, an exception caught as a `Throwable` cannot be simply rethrown. A **`finally`** clause is used to ensure that the method exit is recorded in call history regardless of whether an exception has occurred. The original exception is rethrown without alteration.

When a boundary map affects an existing method, the original method body is moved to a new method named "`__CONTEXT$inner`" followed by a number. The body of the boundary map is placed in a new method named "`__CONTEXT$map`" followed by a number. The original body of the affected method is replaced by a delegating call to the boundary map method. Any calls to proceed within the boundary map are replaced with calls to the new method containing the original method body. Any calls to the affected method that exist within the affected boundary are replaced with calls to the new method containing the original method body. Thus, an invocation of `method1` results in an eventual delegation to the original body of `method2` (now contained in `__CONTEXT$inner1`) and not the new body; this causes the first problem noted in Section 7.

Note that, aside from the **`static`** initializer, inner methods, and map methods, a default constructor has to be added if it is not explicitly declared, in order that it can be instrumented.

B.1 Contents of `icworkingdir/SomeClass.java`

```
public class SomeClass {
  private static CallReceptor __CONTEXT_callReceptor0;
  private static CallReceptor __CONTEXT_callReceptor1;
```

[11] Just after any explicit constructor invocation, if present, in the latter case.

```
Original method or constructor header {
  [Original explicit constructor invocation, if present]
  CallState instantiation
  ReturnState instantiation

  Addition of target instance and arguments to CallState object
  Addition of target instance to ReturnState object
  History._callReceived(CallReceptor for this method,
                        CallState object);

  try {
    Original method body, save that return statements are instrumented
  }
  catch(Throwable __CONTEXT_exception) {
    Recording of exception in ReturnState object
    Testing for each exception type that this method declares it throws
    if(__CONTEXT_exception instanceof RuntimeException)
      throw (RuntimeException)__CONTEXT_exception;
    throw (Error)__CONTEXT_exception;
  }
  finally {
    History._returnFromCall(CallReceptor for this method,
                            ReturnState object);
  }
}
```

Figure 13: General structure of instrumentation for methods and constructors.

```
static {
  __CONTEXT_callReceptor1 =
    new CallReceptor(SomeClass.class,
                     "main",
                     new String[] {"String[]"},
                     new String[] {"args"});
  __CONTEXT_callReceptor0 =
    new CallReceptor(SomeClass.class,
                     "SomeClass",
                     new String[] {},
                     new String[] {});
}

SomeClass() {
  final CallState __CONTEXT_callState0 = new CallState(0);
  final ReturnState __CONTEXT_returnState0 = new ReturnState();

  __CONTEXT_returnState0.setThis(this);
  History._callReceived(__CONTEXT_callReceptor0,
                        __CONTEXT_callState0);

  try {
  }
  catch(Throwable __CONTEXT_exception) {
    __CONTEXT_returnState0.setException(__CONTEXT_exception);
    if(__CONTEXT_exception instanceof RuntimeException)
      throw (RuntimeException)__CONTEXT_exception;
```

```
      throw (Error)__CONTEXT_exception;
    }
    finally {
      History._returnFromCall(__CONTEXT_callReceptor0,
                              __CONTEXT_returnState0);
    }
  }

  public static void main(String[] args) {
    final CallState __CONTEXT_callState1 = new CallState(1);
    final ReturnState __CONTEXT_returnState1 = new ReturnState();

    __CONTEXT_callState1.addParameter(args);
    History._callReceived(__CONTEXT_callReceptor1,
                          __CONTEXT_callState1);

    try {
      OtherClass obj = new OtherClass();
      {
        int i = 0;
        for(; i < 3; i++)
          obj.method1();
      }
    }
    catch(Throwable __CONTEXT_exception) {
      __CONTEXT_returnState1.setException(__CONTEXT_exception);
      if(__CONTEXT_exception instanceof RuntimeException)
        throw (RuntimeException)__CONTEXT_exception;
      throw (Error)__CONTEXT_exception;
    }
    finally {
      History._returnFromCall(__CONTEXT_callReceptor1,
                              __CONTEXT_returnState1);
    }
  }
}
```

B.2 Contents of `icworkingdir/OtherClass.java`

```
public class OtherClass {
  private static CallReceptor __CONTEXT_callReceptor0;
  private static CallReceptor __CONTEXT_callReceptor1;
  private static CallReceptor __CONTEXT_callReceptor2;
  private static CallReceptor __CONTEXT_callReceptor3;
  private static CallReceptor __CONTEXT_callReceptor4;
  private static CallReceptor __CONTEXT_callReceptor5;
  private static CallReceptor __CONTEXT_callReceptor6;

  static {
    __CONTEXT_callReceptor6 =
      new CallReceptor(OtherClass.class,
                       "__CONTEXT$map1",
                       new String[] {},
                       new String[] {});
    __CONTEXT_callReceptor5 =
      new CallReceptor(OtherClass.class,
                       "__CONTEXT$inner1",
                       new String[] {},
                       new String[] {});
    __CONTEXT_callReceptor4 =
      new CallReceptor(OtherClass.class,
                       "__CONTEXT$map0",
                       new String[] {},
                       new String[] {});
    __CONTEXT_callReceptor3 =
      new CallReceptor(OtherClass.class,
                       "__CONTEXT$inner0",
                       new String[] {},
                       new String[] {});
    __CONTEXT_callReceptor2 =
      new CallReceptor(OtherClass.class,
                       "method2",
                       new String[] {},
                       new String[] {});
    __CONTEXT_callReceptor1 =
      new CallReceptor(OtherClass.class,
                       "method1",
                       new String[] {},
                       new String[] {});
```

```
  __CONTEXT_callReceptor0 =
    new CallReceptor(OtherClass.class,
                     "OtherClass",
                     new String[] {},
                     new String[] {});
}

OtherClass() {
  final CallState __CONTEXT_callState0 = new CallState(0);
  final ReturnState __CONTEXT_returnState0 = new ReturnState();

  __CONTEXT_returnState0.setThis(this);
  History._callReceived(__CONTEXT_callReceptor0,
                        __CONTEXT_callState0);

  try{
  }
  catch(Throwable __CONTEXT_exception) {
    __CONTEXT_returnState0.setException(__CONTEXT_exception);
    if(__CONTEXT_exception instanceof RuntimeException)
      throw (RuntimeException)__CONTEXT_exception;
    throw (Error)__CONTEXT_exception;
  }
  finally {
    History._returnFromCall(__CONTEXT_callReceptor0,
                            __CONTEXT_returnState0);
  }
}

public void method1() {
  final CallState __CONTEXT_callState1 = new CallState(0);
  final ReturnState __CONTEXT_returnState1 = new ReturnState();

  __CONTEXT_callState1.setThis(this);
  __CONTEXT_returnState1.setThis(this);
  History._callReceived(__CONTEXT_callReceptor1,
                        __CONTEXT_callState1);

  try {
    __CONTEXT$map0();
  }
```

```
  catch(Throwable __CONTEXT_exception) {
    __CONTEXT_returnState1.setException(__CONTEXT_exception);
    if(__CONTEXT_exception instanceof RuntimeException)
      throw (RuntimeException)__CONTEXT_exception;
    throw (Error)__CONTEXT_exception;
  }
  finally {
    History._returnFromCall(__CONTEXT_callReceptor1,
                            __CONTEXT_returnState1);
  }
}

public void method2() {
  final CallState __CONTEXT_callState2 = new CallState(0);
  final ReturnState __CONTEXT_returnState2 = new ReturnState();

  __CONTEXT_callState2.setThis(this);
  __CONTEXT_returnState2.setThis(this);
  History._callReceived(__CONTEXT_callReceptor2,
                        __CONTEXT_callState2);

  try{
    __CONTEXT$map1();
  }
  catch(Throwable __CONTEXT_exception) {
    __CONTEXT_returnState2.setException(__CONTEXT_exception);
    if(__CONTEXT_exception instanceof RuntimeException)
      throw (RuntimeException)__CONTEXT_exception;
    throw (Error)__CONTEXT_exception;
  }
  finally {
    History._returnFromCall(__CONTEXT_callReceptor2,
                            __CONTEXT_returnState2);
  }
}

private void __CONTEXT$inner0() {
  final CallState __CONTEXT_callState3 = new CallState(0);
  final ReturnState __CONTEXT_returnState3 = new ReturnState();

  __CONTEXT_callState3.setThis(this);
  __CONTEXT_returnState3.setThis(this);
  History._callReceived(__CONTEXT_callReceptor3,
                        __CONTEXT_callState3);

  try {
    __CONTEXT$inner1();
  }
  catch(Throwable __CONTEXT_exception) {
    __CONTEXT_returnState3. setException(__CONTEXT_exception);
    if(__CONTEXT_exception instanceof RuntimeException)
```

```
      throw (RuntimeException)__CONTEXT_exception;
    throw (Error)__CONTEXT_exception;
  }
  finally{
    History._returnFromCall(__CONTEXT_callReceptor3,
                            __CONTEXT_returnState3);
  }
}

private void __CONTEXT$map0() {
  final CallState __CONTEXT_callState4 = new CallState(0);
  final ReturnState __CONTEXT_returnState4 = new ReturnState();

  __CONTEXT_callState4.setThis(this);
  __CONTEXT_returnState4.setThis(this);
  History._callReceived(__CONTEXT_callReceptor4,
                        __CONTEXT_callState4);

  try {
    Call call = History.lastCall(null, "method2", null);
    if(call == null)
      __CONTEXT$inner0();
    else
      System.err.println("method2() called before");

    Call call2 =
      History.lastCall(String.class, "length", null);
    System.err.println(call2 != null);
  }
  catch(Throwable __CONTEXT_exception) {
    __CONTEXT_returnState4.setException(__CONTEXT_exception);
    if(__CONTEXT_exception instanceof RuntimeException)
      throw (RuntimeException)__CONTEXT_exception;
```

```
    throw (Error)__CONTEXT_exception;
  }
  finally{
    History._returnFromCall(__CONTEXT_callReceptor4,
                            __CONTEXT_returnState4);
  }
}

private void __CONTEXT$inner1() {
  final CallState __CONTEXT_callState5 = new CallState(0);
  final ReturnState __CONTEXT_returnState5 = new ReturnState();

  __CONTEXT_callState5.setThis(this);
  __CONTEXT_returnState5.setThis(this);
  History._callReceived(__CONTEXT_callReceptor5,
                        __CONTEXT_callState5);

  try {
    String s = "a string";
    System.err.println(s.length());
  }
  catch(Throwable __CONTEXT_exception) {
    __CONTEXT_returnState5.setException(__CONTEXT_exception);
    if(__CONTEXT_exception instanceof RuntimeException)
      throw (RuntimeException)__CONTEXT_exception;
    throw (Error)__CONTEXT_exception;
  }
  finally{
    History._returnFromCall(__CONTEXT_callReceptor5,
                            __CONTEXT_returnState5);
  }
}

private void __CONTEXT$map1() {
  final CallState __CONTEXT_callState6 = new CallState(0);
  final ReturnState __CONTEXT_returnState6 = new ReturnState();

  __CONTEXT_callState6.setThis(this);
  __CONTEXT_returnState6.setThis(this);
  History._callReceived(__CONTEXT_callReceptor6,
                        __CONTEXT_callState6);
```

```
    try {
      __CONTEXT$inner1();
    }
    catch(Throwable __CONTEXT_exception) {
      __CONTEXT_returnState6.setException(__CONTEXT_exception);
      if(__CONTEXT_exception instanceof RuntimeException)
        throw (RuntimeException)__CONTEXT_exception;
      throw (Error)__CONTEXT_exception;
    }
    finally {
      History._returnFromCall(__CONTEXT_callReceptor6,
                              __CONTEXT_returnState6);
    }
  }
}
```

C API to Communication History

The implementation of the query interface to communication history is described at a conceptual level in Section 5.5. Here is described its application programming interface (API). Section C.1 describes the API to the `History` class and Section C.2 describes the API to the `Call` class.

C.1 The `History` Class

The API for the `History` class consists of seven query methods. These are described on the following pages as indicated in the table below. All these methods are internally synchronized to operate in a multithreaded environment.

Query method	page(s)
`firstCall`	45
`lastCall`	48
`lastCallAnySubclass` (2 forms)	46, 47
`lastCallInCFlow`	49
`lastCallReturningClass`	46
`lastInstancePassed`	46
`mostRecentCall`	46

firstCall

```
public static Call firstCall(Thread thread,
                             Class cls,
                             String mth,
                             String[] paramTypes,
                             Object returnVal,
                             Call relativeTo)
```

Searches for and returns the earliest call conforming to the search parameters passed as arguments; returns **null** if no call can be found that matches these parameters.

Parameter	*Description*	*Default*
`thread`	The thread of control whose tree should be searched, in the case of causal searches; or **null** if temporal searches should be performed	current thread
`cls`	The exact class on which the executed method was defined; or **null** if the class is not of interest	—
`mth`	The name of the method that was executed; or **null** if the method is not of interest	**null**
`paramTypes`	An array containing the names of the types of the formal parameters as they appear lexically in the method declaration; or **null** if the parameter types are not important	**null**
`returnVal`	The object that must have been returned by the method execution; or **null** if the returned object is not of interest	**null**
`relativeTo`	The call at which the search should begin; the predecessors of this call are compared, in temporal order, to the search parameters	`mostRecentCall()`

mostRecentCall

public static `Call mostRecentCall()`

Searches for and returns the most recently occurring call of any form.

Parameter	*Description*	*Default*
Takes no parameters		

lastInstancePassed

public static `Object lastInstancePassed(Class cls)`

Searches for and returns the object of type `cls` that was most recently passed as a target or argument to some call; returns **`null`** if no such call can be found.

Parameter	*Description*	*Default*
`cls`	The class of which type to search for the most recently passed instance	—

lastCallReturningClass

public static `Call lastCallReturningClass(Class cls)`

Searches for and returns the most recently occurring call that returns an object of a type that is assignable to the class passed in `cls`; returns **`null`** if no call can be found that matches these parameters.

Parameter	*Description*	*Default*
`cls`	The class to which the type of the returned object is assignable; or **`null`** if the class is not of interest	—

lastCallAnySubclass

public static `Call lastCallAnySubclass(Class cls)`

Searches for and returns the most recently occurring call on any class that is assignable to the class passed in `cls`; returns **`null`** if no call can be found that matches these parameters.

Parameter	*Description*	*Default*
`cls`	Some superclass of the class on which the executed method was defined; or **`null`** if the class is not of interest	—

lastCallAnySubclass

```
public static Call lastCallAnySubclass(Thread thread,
                                       Class cls,
                                       String mth,
                                       String[] paramTypes,
                                       Object returnVal,
                                       Call relativeTo)
```

Searches for and returns the most recently occurring call on any class that is assignable to the class passed in `cls`; returns **`null`** if no call can be found that matches these parameters.

Parameter	*Description*	*Default*
`thread`	The thread of control whose tree should be searched, in the case of causal searches; or **`null`** if temporal searches should be performed	—
`cls`	Some superclass of the class on which the executed method was defined; or **`null`** if the class is not of interest	—
`mth`	The name of the method that was executed; or **`null`** if the method is not of interest	—
`paramTypes`	An array containing the names of the types of the formal parameters as they appear lexically in the method declaration; or **`null`** if the parameter types are not important	—
`returnVal`	The object that must have been returned by the method execution; or **`null`** if the returned object is not of interest	—
`relativeTo`	The call at which the search should begin; the predecessors of this call are compared, in temporal order, to the search parameters	—

lastCall

```
public static Call lastCall(Thread thread,
                            Class cls,
                            String mth,
                            String[] paramTypes,
                            Object returnVal,
                            Call relativeTo)
```

Searches for and returns the most recently occurring call conforming to the search parameters passed as arguments; returns **null** if no call can be found that matches these parameters.

Parameter	*Description*	*Default*
thread	The thread of control whose tree should be searched, in the case of causal searches; or **null** if temporal searches should be performed	current thread
cls	The exact class on which the executed method was defined; or **null** if the class is not of interest	—
mth	The name of the method that was executed; or **null** if the method is not of interest	—
paramTypes	An array containing the names of the types of the formal parameters as they appear lexically in the method declaration; or **null** if the parameter types are not important	—
returnVal	The object that must have been returned by the method execution; or **null** if the returned object is not of interest	**null**
relativeTo	The call at which the search should begin; the predecessors of this call are compared, in temporal order, to the search parameters	mostRecentCall()

lastCallInCFlow

```
public static Call lastCallInCFlow(Thread thread,
                                   Class cls,
                                   String mth,
                                   String[] paramTypes,
                                   Object returnVal,
                                   Call cflowOf)
```

Searches for and returns the most recently occurring call that occurred as a child of the call in `cflowOf`, also conforming to the other search parameters passed as arguments; returns **`null`** if no call can be found that matches these parameters.

Parameter	*Description*	*Default*
`thread`	The thread of control whose tree should be searched, in the case of causal searches; or **`null`** if temporal searches should be performed	—
`cls`	The exact class on which the executed method was defined; or **`null`** if the class is not of interest	—
`mth`	The name of the method that was executed; or **`null`** if the method is not of interest	—
`paramTypes`	An array containing the names of the types of the formal parameters as they appear lexically in the method declaration; or **`null`** if the parameter types are not important	—
`returnVal`	The object that must have been returned by the method execution; or **`null`** if the returned object is not of interest	—
`cflowOf`	The call whose children are to be searched for a match	—

C.2 The `Call` Class

The `History` class possesses a helper class, called `Call`, that provides an interface to the values related to specific calls.

getMethodName

```
public java.lang.String getMethodName()
```

Returns the name of the method that was invoked by this call, as determined by the tool's instrumentation (see Appendix B).

getThread

```
public java.lang.Thread getThread()
```

Returns the thread instance in which this call was invoked.

getReturnValue

```
public java.lang.Object getReturnValue()
```

Returns the result value returned by this call, as an `Object` (meaning that primitives are wrapped in instances of their corresponding classes), or **`null`** if the call has not returned or has a result type of **`void`**.

getTarget

```
public java.lang.Object getTarget()
```

Returns the instance upon which this call was invoked, or **`null`** if the invoked method was **`static`**.

getTargetClass

```
public java.lang.Class getTargetClass()
```

Returns the class upon which the method was declared that was invoked by this call.

getParameter

```
public java.lang.Object getParameter(int i)
```

Returns the `i`th argument that was passed in this call, as an `Object` (meaning that primitives are wrapped in instances of their corresponding classes).

getPredecessor

```
public Call getPredecessor()
```

Returns the call that temporally preceded this one.

getParent

```
public Call getParent()
```

Returns the call that invoked the method that sent this call.

precedes

```
public boolean precedes(Call other)
```

Determines whether this call temporally precedes the call passed in `other`.