Combining Runtime and Static Universe Type Inference

Andreas Fürer

Master Project Report

Software Component Technology Group
Department of Computer Science
ETH Zurich

http://sct.inf.ethz.ch/

September 11, 2006 – March 10, 2007

Supervised by:
Dipl.-Ing. Werner M. Dietl
Prof. Dr. Peter Müller

Software Component Technology Group
inf | Informatik
Computer Science

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Abstract

The Universe type system provides means to structure the heap memory. This structuring enables to reason about object structures. Annotating Java source code with Universe type modifiers can be a big effort for developers. To ease the manual annotating tasks, we use inference of Universe type modifiers. Preceding projects worked on two different inference approaches: runtime inference, where execution traces are used to create an extended object graph and infer types, and static inference, where a SAT solver is used to find correct modifiers. In this thesis, we work on combining the two approaches and enhancing the tools. We use the runtime inference to define a weight scheme that is respected by the static inference. In such a way we can use the strengths of both inference approaches. We implemented the tools in the Eclipse IDE and improved the usability of the Universe tools. The tools are integrated as a set of plug-ins which enable intuitive configuration and interaction with the inferer. Type modifiers can directly be added to the source code. We also worked on the graphical visualization of Universe structures.

Acknowledgments

I would like to thank my supervisor Werner Dietl for all his contributions to this thesis and the useful advices on the report. Also, I would like to thank Prof. Peter Müller and the whole SCT group for the good and friendly atmosphere.

Special thanks to my parents for supporting me throughout my studies at the ETH. Without your support that would not have been possible.

Finally, thanks to my girlfriend Natalie.

Contents

Glossary

0.1. General Terms and Abbreviations

AST Abstract Syntax Tree

EOG Extended Object Graph

JDK Java Development Kit

JML Java Modeling Language

JSR Java Specification Request

JVM Java Virtual Machine

JVMTI JVM Tool Interface

MAX-SAT Maximum Satisfiability Problem

MJ MultiJava

PBS Pseudo Boolean Solver

RI Runtime Inference

SI Static Inference

SUTI Static Universe Type Inference

UI User Interface

XSD XML Schema Definition

0.2. Eclipse Specific Terms and Abbreviations

Editor Editors are used to view and edit a specific resource (e.g., Java file, XML file, C++ file, ...). Editors follow the common open-save-close model.

EMF Eclipse Modeling Framework

Feature A feature is a unit of download and installation consisting of one or more plug-ins. It is used to easier package and deploy several Eclipse plug-ins.

GEF Graphical Editing Framework, an Eclipse framework to create rich graphical editors.

GMF Graphical Modeling Framework

JDT Java Development Tools

PDE Plugin Development Environment

Perspective A perspective is a set of views. It groups views with related tasks (e.g., Java Perspective, C++ Perspective, Debug Perspective, CVS Perspective).

RCP Rich Client Platform. The concept of using the platform functionality (e.g., UI, plug-in mechanism) of Eclipse as a framework for other programs.

SWT Standard Widget Toolkit, an open source framework for developing graphical user interfaces in Java. The Eclipse alternative for AWT or Swing.

View A view is a window within a workbench page. Views are typically used to navigate resources and modify properties of a resource (e.g., Package Explorer, Outline View, Problems View). Only a single instance of any one view can be open at one time.

Workbench Workbench is the name of the user interface infrastructure in Eclipse. It is responsible for the presentation and coordination of the user interface. Internally, the workbench is a single root object that contains the menu bar, tool bar, perspective bar, status line, and a main area for displaying pages containing workbench parts. Only one page can be active at a time.

Workbench page A page can contain zero or more workbench parts.

Workbench part A workbench part can be a view or an editor.

Workspace The workspace is a folder on the disk. All project settings and files are stored within a workspace.

Chapter 1.

Introduction

1.1. Motivation

Object-oriented programming languages access objects through references. An effect of this mechanism is aliasing. An alias occurs if two ore more variables hold a reference to the same object, that is, they point to the same memory cell. In some cases aliasing is intended for efficiency reasons (e.g., pass-by-reference), but there are unintended drawbacks of aliasing as well. Concepts and techniques like information hiding and object encapsulation do not prevent *leaking* and *capturing* of references. Reference leaking occurs when a data structure passes a reference to an object, which is supposed to be internal, to the outside. Reference capturing occurs when objects are passed to a data structure and then stored by the data structure.

The Universe type system[28] provides a mechanism for aliasing control. It structures the heap and enforces encapsulation of whole object structures to enable modular verification and reasoning about complex programs.

Manually structuring large programs with Universe types can be a big effort. The motivation of the Universe type inference is to automatically inferring the Universe structure. That would ease the task of a programmer. Two different inference approaches are feasible: a runtime observation of a program (Runtime Inference) or extraction of the type relations from the source code statically (Static Inference).

1.2. Universe Type System

The Universe type system structures the heap into Universes. It is an ownership type system that assigns at most one owner to each object. Objects with the same owner are grouped into one context (called "Universe"). Objects without owner are in the root context which forms the root of the ownership hierarchy. Only references *within* a context and *from an owner to an owned object* can be used for modifications, either by field updates or method calls. Other references can only be used to read fields and call side-effect-free methods. The ownership properties are expressed with ownership modifiers that are used to annotate each reference:

- **peer** denotes a reference to an object inside the same context.
- **rep** denotes a reference from an object into the context it owns.
- **readonly** denotes a reference that is read-only and might point to objects in any arbitrary context.

Since primitive types are value and not reference types, they cannot be aliased and do not need Universe annotations. Notice that the Generics Universe Type formalization [11] introduced the **any** modifier instead of the **readonly** modifier. To be backward-compatible with the JML implementation[12], we still use the **readonly** keyword.

Methods can be annotated with the modifier **pure** if they do not modify existing objects.

Type Combinator

In order to choose Universe types of transitive accesses like an attribute access, array accesses, or method invocations, the type of the expression is determined by the type combinator function $\star$. The type combinator takes two ownership modifiers and returns the resulting ownership modifier (see Table 1.1). For the field access `x.f`, `x` is the first argument (left-most cells of the rows) and `f` the second argument (top-most cells of the columns). Dereferencing chains are applied with a left-right preference. Analogous rules are used for method invocations.

$\star$	**peer**	**rep**	**readonly**
peer	peer	readonly	readonly
rep	rep	readonly	readonly
readonly	readonly	readonly	readonly

Table 1.1.: Type combinator of the Universe types. The `this` reference is always a **peer** reference. If the first argument is a `this` reference, the type combinator is not applied.

Subtyping

The subtype relation on Universe types follows the subtype relation in Java: two **peer**, **rep**, or **readonly** types are subtypes if the corresponding classes or interfaces are subtypes in Java. Furthermore, every **peer** and **rep** type is a subtype of the **readonly** type with the same class.

Tool Support

The Universe type system is integrated into the MultiJava[9] compiler and the Java Modeling Language (JML) Tools [13, 12]. A JML tool integration into Eclipse was developped in [5] and our current work is a further step heading in the same direction.

1.3. Runtime Inference Tool

Two preceding Master projects dealt with the topic of Runtime Inference. These were the works done by Frank Lyner[24] and Marco Bär[4]. This section will outline the ideas implemented in these projects.

1.3.1. Inference Algorithm Overview

The inference algorithm consists of the following steps:

1. Program monitoring with a tracing agent.
2. Building the datastructure.
3. Structuring the object store in Universe contexts.
4. Finding valid annotations.
5. Generating output into an annotation XML file.

The program monitoring is done with a tracing agent which runs as an agent in the Java Virtual Machine and gets notified about the inspected program using the Java Virtual Machine Tool Interface (JVMTI). The gathered tracing information is stored in a trace XML file. The type inferer parses the tracing file and builds up the Extended Object Graph (EOG), the datastructure that is used for the type inference. An EOG is a representation of the heap during the whole execution time. Nodes of the EOG represent the objects on the heap; vertices are the references. We distinguish between two kinds of references: the first are *variable references* used to denote relationships between two objects and the associated variable that stores the reference. The latter are *write references* used to express a write operation between two objects. Write references are not associated with a variable. Method calls of non-pure methods also introduce write references. Once the graph is built up, the type inferer assigns an owner to each node in the EOG using an *owner-as-dominator* algorithm as an approximation for the Universe structure. Possible conflicts like write references that cross context boundaries are resolved in the *conflict resolution* and *harmonization* phase and, finally, establish the *owner-as-modifier* property. During the annotation phase every variable reference in the graph is followed and the ownership modifiers are determined as a result of the Universe structure. The annotation structure is externalized in an XML file conforming to the annotations.xsd schema (see Appendix C).

1.3.2. Static Methods, Arrays, and Method Body Annotations

The scope of Frank Lyner's work was limited to annotations of fields and method signatures. Static method calls, arrays, and annotations of method bodies (i.e., local variables, object creations, casts) were not addressed. To bring the Runtime Inference to a practical level, the work done by Marco Bär resolved these issues.

Static Methods and Object Creations

Static methods and objects do not have a target because they are not related to a specific instance of a class. This implies that **rep** types must not be used in signatures of static methods or anywhere else inside a method. Bär extended the EOG with an artifical target object for each static method call. The extension of the object graph enabled to annotate static method calls and object creations.

Arrays

Array operations do not trigger the standard events defined by the JVMTI, therefore, array handling was ignored in [24]. Bär used the possibility of bytecode instrumentation to generate the missing events for array operations. In the Universe type system, references to arrays of reference types need two ownership modifiers: the first describes the relationship between the `this` object and the array, the second the relationship between the array and the objects referenced by its components. In order to be able to handle array component updates, Bär introduced a new kind of reference to the EOG, called *component references*, that connect array objects with the objects referenced by its components.

Annotations of Method Bodies

Annotations of method bodies can lead to several problems like the *dereferencing chains* problem or *bad code coverage* issues that are discussed in Section 2.2. A dereferencing chain is a number of read accesses followed by a write access. Dereferencing chains are problematic because the JVMTI tracing agent only generates an event for the write access to the object on the stack at last.

Bär's approach to annotate method bodies relies on an *abstract interpretation*. The abstract interpretation which he performs is very similar to the one that bytecode verifiers for the JVM prior to Java 6 implement.

1.4. Static Inference Tool

Two preceding Master projects dealt with the topic of the Static Inference. These were the works done by Nathalie Kellenberger[22] and Matthias Niklaus[29]. In this section we will outline the ideas implemented in these projects.

1.4.1. Inference Algorithm Overview

The inference algorithm consists of the following steps:

1. Source code parsing.
2. Generation of constraints out of the syntax tree.
3. Finding solutions.
4. Generating output into an annotation XML file.

The source code parsing is implemented using the MultiJava[9] and JML[13] parser. Out of the Abstract Syntax Tree(AST), the constraints describing the Universe type system properties are generated and written down in a designated format. An external solver is used to determine the inference solution of the Universe types. The inference result is written to an annotation XML file with the same format that is used for the Runtime Inference (see Appendix C).

1.4.2. Problem Solver

Natalie Kellenberger used Prolog as the solver system. That led to an implementation which was tightly coupled to Prolog and was not very practical because Prolog could not find inference solutions for overconstrained systems. Also the restrictions on Universe type casts were too strong.

Matthias Niklaus introduced an abstract interface between the Java front-end and an inferer back-end, called the UTI interface. This abstraction enables to support different inferer back-ends. Because our Universe type inference problem is an optimization and not a decision problem, we are more interested in the formulation of a MAX-SAT problem than using Prolog. Niklaus has chosen the pseudo boolean solver PBS v2.1[3] as the problem solver.

Using a MAX-SAT solver enables to define a qualitative description of the type inference solution by setting weights. More details are explained in Section 2.6.

1.5. Goals and Requirements

The goal of this thesis is to combine runtime and static Universe type inference. The existing tools have to be enhanced in a way that they can be used together. We think that the inference result can be improved by combining the two inference approaches.

The Runtime and Static Inference tools described above are both stand-alone tools. The configuration and usage of these tools can be quite complicated. Our goal is to integrate them as a collection of plug-ins in the Eclipse IDE. We integrate the Universe inference tools together with the JML tools implementation done in the semester project by Paolo Bazzi[5]. The integration in Eclipse is crucial to combine and simplify the usage of the different tools. Together with a visualization of the Universe structure, we think that we can improve the usability of the Universe type system tools and make it available to a wider community of developers. We integrate the tools in a way that we do not break the existing functionality of the programs and it is still possible to use the tools on the command line without Eclipse.

In Chapter 2 we present our ideas about combining the two inference approaches. Chapter 3 is the user guide of the Eclipse plug-ins that we have developed. The implementation is outlined in Chapter 4. Finally, Chapter 5 concludes this thesis with the discussion of the results that we have achieved.

Chapter 2.

Combining Runtime and Static Inference

In this chapter we will explain how we combine the Static and the Runtime Inference. We start with a conceptual comparison of the inference approaches, showing the pros and cons in Section 2.1. We deepen the comparison with an assessment in Section 2.2 and 2.3 where we look at some specific examples. In Section 2.4 we discuss some combination concepts. Furthermore, we outline some ideas based on the results of our assessment in 2.5 and show the way how we solve it in our current implementation in Section 2.6.

2.1. Comparison of Runtime and Static Inference

We start with a comparison of Runtime and Static Inference. Let us consider an overview of the assets and drawbacks of both inference concepts.

2.1.1. Runtime Inference

Runtime Inference delivers good inference quality provided that the code coverage is good. The following items list some properties that we observed:

1. *Solution Quality*
   The Runtime Inference always finds the solution with the deepest nested Universe structure. However, good code coverage is crucial for the solution quality. Infering types for test cases with bad code coverage might lead to bad annotations. Also, a particular runtime trace does not generalize to future executions of a program.
   The annotation of super declarations of methods is not supported yet. Methods declared in interfaces and methods of abstract classes are not annotated at all. The JVMTI events are only triggered for the implementing classes. Although the annotations are inferred for the class that triggered the event, they are not propagated to the superclasses and interfaces during the harmonization phase of the inference algorithm.

2. *Method Body Annotations*
   Unfortunately, it is not possible to generate events for modifications of local variables in JVMTI. Therefore, the Runtime Inference cannot annotate method bodies. The approach taken by [4], the abstract interpretation of the bytecode, was a static analysis and is not a pure runtime analysis approach.

3. *Partially Annotated Programs*
   The Universe annotations in a program do not trigger a JVMTI event if runtime checks are disabled. It is not possible that a programmer introduces some annotations that are assumed to be correct and let the inferer annotate the rest of the program.

4. *Time Consumption*
   Because the Runtime Inference monitors the program execution, we need at least the running time of the actual program. Examples that generate huge tracing files lead to a bottleneck during the parsing of the tracing file and the build-up phase of the Extended Object Graph (EOG).

2.1.2. Static Inference

The solution quality of the Static Inference depends on the preference settings and is not always optimal. On the other hand it allows to annotate method bodies and partially annotated programs. Let us consider the following points.

1. *Solution Quality*
   We cannot assure that the solution returned first by the PBS solver is the optimal solution. The solution quality depends on the settings of the preferences (see Section 2.6). But contrary to the Runtime Inference, the developer can interact with the inferer and influence the solution quality.

2. *Method Body Annotations*
   Since the Static Inference parses the source code, we have all required information to build constraints about method bodies like local variable declarations. We do not need a special concept for the annotation of method bodies.

3. *Partially Annotated Programs*
   Annotations can be added directly in the source code and are parsed by the JML parser. It is possible, therefore, to use partially annotated programs. The given annotations are inserted in the generated constraints.

4. *Time Consumption*
   For small examples the performance of the Static Inference is good and is faster than the Runtime Inference. Nevertheless, the MAX-SAT problem has a high computational complexity and is NP-hard for arbitrary propositional logic expression. Also, using an approximation algorithm for the MAX-SAT problem is a complex problem as it is stated in the following theorem:

   There is no polynomial-time approximation scheme for MAX-SAT unless $P = NP$. [6]

   Even though our MAX-SAT solver has good performance, there can be cases where running time is exponential.

5. *Bad Casts*
   We cannot guarantee that all casts in a program work out during runtime. It is possible that casts are statically correct and compilable, but fail at runtime.

2.2. Assessment of Runtime Inference

As we mentioned earlier we want to use the the Runtime Inference in its original form without the method body annotation which actually is a form of static inference. We strictly want to inspect the runtime behavior of a program and infer the missing parts in the Static Inference phase.

2.2.1. Bad Code Coverage

There are cases where the Runtime Inference leads to undesired annotations. These annotations are good in a certain test case, but having the knowledge of the whole code base it seems obvious that there are more suitable Universe annotations. Let us look at this problem by considering an example.

Listing 2.1: A bad code coverage example.

```
 1  public class Coverage {
 2      Coverage field1;
 3      Coverage field2;
 4
 5      public void foo(boolean condition) {
 6          field1 = new Coverage();
 7          field2 = new Coverage();
 8          if (condition) {
 9              field1.makePeer(this);
10          } else {
11              field2.makePeer(this);
12          }
13      }
14
15      public void doWrite() {
16          // creates a write reference
17      }
18
19      public void makePeer(Coverage other) {
20          other.doWrite();
21      }
22
23      public static void main(String[] args) {
24          Coverage c = new Coverage();
25          c.foo(true); // or c.foo(false) for other path
26      }
27
28  }
```

Since the Runtime Inference is based on the runtime behavior of a program, our Universe type annotations depend on a given test run. This means that we may have execution paths that might not have been taken by the program, leading to wrong annotations. Even worse, it can lead to code that is not compilable and rejected by the JML compiler. Consider the example in Listing 2.1: there are two execution paths depending on the case of the predicate `condition` in the if expression. In a given test case we may set the `condition` to true. The Runtime Inference algorithm will annotate `field1` with **peer** as follows: the first invocation of `makePeer(Coverage)` on line 9 inserts a *write reference* from `this` to the `Coverage field1`.

The call of the non-**pure** method `other.doWrite()` on line 20 inserts a second *write reference* from the target object of the method call (the `field1`) to the `this` object. This cycle of write references enforces the **peer** annotation.

The `field2` remains untouched and, therefore, is annotated as **rep**. However, if we try to compile the code with these annotations, we will get an error. The method call `field2 .makePeer(this)` on line 11 is illegal because there is no way to cast a **rep** object to **peer**. Considering the opposite test case with a parameter `false` would lead to the same situation with a **peer** annotation for `field2` and **rep** for `field1`. The reason for this uncompilable example is a bad coverage issue. We need to take into account that test cases that do not cover the whole code can result in wrong inference results.

Listing 2.2: A doubly-linked list example.

```
public class LinkedList {
    /*@ rep @*/ Node first;

    LinkedList() {
        first = null;
    }

    public void createList() {
        Data d1 = new Data(1);
        Data d2 = new Data(2);
        Data d3 = new Data(3);
        Data d4 = new Data(4);
        first = new Node(d1);
        first.insertTail(d2);
        first.insertTail(d3);
        first.insertTail(d4);
    }

    public static void main(String[] args) {
        LinkedList l = new LinkedList();
        l.createList();
    }
}

public class Node {
    /*@ rep @*/ Node next;
    /*@ readonly @*/ Node previous;
    Data item;

    public Node(Data toStore) {
        item = toStore;
        next = null;
    }

35      public void insertTail(Data toStore) {
36          if (next == null) {
37              next = new Node(toStore);
38              next.previous = this;
39          } else {
40              next.insertTail(toStore);
41          }
42      }
43
44      public void insertHead(Data toStore) {
45          if (previous == null) {
46              previous = new Node(toStore);
47              previous.next = this;
48          } else {
49              previous.insertHead(toStore);
50          }
51      }
52  }
53
54  public class Data {
55      private int value;
56
57      public Data(int i) {
58          value = i;
59      }
60
61  }
```

Problems may also arise in situations as described in the Example 2.2. A doubly-linked list is implemented in two classes: the `LinkedList` which implements the list head and the `Node` which represents an element in the doubly-linked list that will hold a reference to a stored object.

New elements can be added at the head or the tail of the list. Notice that not the list head inserts new list items, but a new item inserted with `insertTail(Data)` is handed down the list and appended at the end. Similarly, new elements are moved towards the list head using the method `insertHead(Data)`. Considering the example, we see that all elements are inserted at the list tail. So, we only have *write references* in the Extended Object Graph (EOG) in one direction (from the list head towards the end). The back links to the predecessor items are represented as *variable references*. Hence, the Runtime Inference annotates the field next with **rep** and the field `previous` with **readonly** as it can be seen in Listing 2.2. We would prefer a solution with **peer** annotations for both fields `next` and `previous`. Despite the fact that the program would not compile because of the missing annotations (those for the method bodies and the item field), this inference result is wrong. Consider method `insert(Data)`: the assignment on line 47 is illegal because the target `previous` is **readonly**. The same argument applies for the non-**pure** method call on a **readonly** target in line 49.

As in the example 2.1, this result is originated by bad coverage. The method `insertHead (Data)` is never executed and thus no write access over the previous reference is performed.

That is why the Runtime Inference annotated the `previous` reference with **readonly** instead of **peer**. If we use the outcome of this test case as an input into the Static Inference, we have to take into account that the bad code coverage could lead to not optimal or even wrong ownership annotations.

2.2.2. Bad Method Sequence

The following problem was already discovered in Frank Lyner's report [24], but with a wrong conclusion. A singly-linked list is implemented as shown in Listing 2.3 and is similar to the example in Section 2.2.1: `LinkedList` implements the list head and the `Node` represents a list element. For simplicity reasons the items are only linked in one direction.

Listing 2.3: A singly-linked list example.

```
public class LinkedList {
    /*@ rep @*/ Node first;

    LinkedList() {
        first = null;
    }

    public void createList() {
        Data d1 = new Data(1);
        Data d2 = new Data(2);
        Data d3 = new Data(3);
        Data d4 = new Data(4);
        first = new Node(d1);
        first.insert(d2);
        first.insert(d3);
        first.insert(d4);
        first.remove(d2);
    }

    public static void main(String[] args) {
        LinkedList l = new LinkedList();
        l.createList();
    }
}

public class Node {
    /*@ readonly @*/ Node next;

    Data item;

    public Node(Data toStore) {
        item = toStore;
        next = null;
    }

    public void insert(Data toStore) {
        if ( next == null ) {
            next = new Node(toStore);
        } else {
            next.insert(toStore);
        }
    }

    public Data remove(Data toRemove) {
        if ( next == null ) {
            return null;
        }
        else if ( toRemove.equals(next.item) ) {
            Data ret = next.item;
            next = next.next;
            return ret;
        } else {
            return next.remove(toRemove);
        }
    }
}
```

If elements are only inserted in the list, the Runtime Inference algorithm will find a **rep** annotation for the `next` field. But if an element is deleted from the middle of the list and no new element is inserted, then the predecessor of the deleted element will receive a *variable reference* to the successor in the Extended Object Graph. These two objects will not be connected by a *write reference*. The variable reference will therefore connect two objects that are neither peers nor owner-owned related. The algorithm will have no other choice than to annotate the `next` field with **readonly** in the harmonization phase which is clearly not precise enough and would require downcasts to be correct. As you may have noticed, the example would be rejected by the JML compiler because the call `next.insert(toStore)` is executed on the **readonly** target field `next`. The field has to be cast to **rep** or **peer**. The Runtime Inference cannot detect the relation between the field access and the method call during the JVMTI event callback handling. The target for the method invocation is just an object reference and not related to the field. The resolution of this conflict is handled in the abstract method body analysis which we skipped because we annotate method bodies in the Static Inference.

If, on the other hand, another object is inserted in the list after an object was deleted, then the predecessor would have a *write reference* to the successor. This would result in a false cycle in the Extended Object Graph, which would be resolved by making all the items **peer** to each other. The field `next` would be annotated with **peer**.

As we can see, the imprecise annotations are not triggered by bad code coverage. Both methods `insert(Data)` and `remove(Data)` are invoked and the field `first` is covered. We have a code coverage of 100%. Also a path coverage of 100% would not help. There are still possible sequences of method invocations that lead to a non-optimal Universe inference result.

2.3. Assessment of Static Inference

Let us consider the example in Listing 2.4 where the Static Inference does not infer an optimal Universe structure:

We have a car with five wheels, four normal wheels and one spare wheel. A wheel can be deflated which sets the tire pressure to the ambient pressure of 1 bar. If the pressure of a tire falls below 2 bar, the car is not safe anymore and must not be used. Of course, this does only apply to the normal wheels and not to the spare wheel. Although it is not recommendable to drive with a deflated spare wheel, that does not harm the safety of the car. To prevent deflating a wheel, we set the preferences for the Static Inference in a way that fields get annotated with **readonly** for the declared type and **rep** for the runtime type. We use **readonly** for the declared type in order to prevent unintended deflating of a tire.

Listing 2.4: Universes for the automobile roadway repair service

```
public class Wheel {
    private float pressure;

    public void deflate() {
        pressure = 1.0f;
    }
}

public class Car {
    /*@ readonly @*/ Wheel frontLeft;
    /*@ readonly @*/ Wheel frontRight;
    /*@ readonly @*/ Wheel rearLeft;
    /*@ readonly @*/ Wheel rearRight;
    /*@ readonly @*/ Wheel spareWheel;

    public Car() {
        frontLeft = new /*@ rep @*/ Wheel();
        frontRight = new /*@ rep @*/ Wheel();
        rearLeft  = new /*@ rep @*/ Wheel();
        rearRight = new /*@ rep @*/ Wheel();
        spareWheel = new /*@ rep @*/ Wheel();
    }
}

public class Driver {
    private /*@ readonly @*/ Car car;

    public Driver() {
        car = new /* peer */Car();
    }

    public void doJob() {
        /*@ readonly @*/ Wheel sw = car.spareWheel;
        ((/*@ peer @*/Wheel)sw).deflate();
    }

    public static void main(/*@ readonly readonly @*/ String[] args) {
        /*@ peer @*/ Driver me = new /*@ peer @*/ Driver();
        me.doJob();
    }
}
```

For some unknown reasons the driver decides to deflate the spare wheel. Therefore, the Static Inference inserts a cast for the spare wheel to **peer**. It is cast to **peer** because the preferences are set in such a way that local variables are annotated with this type. We used the Extended Conflict Type system with a 6 bit encoding for this example. Statically, this cast is correct, but fails at runtime because the dynamic **rep** type cannot be cast to **peer**.

This example shows that the annotation of the Static Inference depend on the preference settings. The quality of the inference solution is just as good as the preferences are set. Furthermore, it shows that the Static Inference can introduce bad cast. Statically, they are correct and can be type checked, but fail at runtime.

2.4. Approaches

In this section we present some approaches for how to combine Runtime and Static inference in a high-level view.

2.4.1. Partially Annotated Sources

In a first step we run the Runtime Inference and insert the inferred solution with the annotator directly into the Java source code. Because the Static Inference is using the JML parser, we are able to parse and interpret the Universe modifiers correctly. This approach is a simple combination with a straightforward implementation.

The downside of this approach is the too strong restriction of the solution domain for the Static Inference. The annotations in the source code set fixed types for some Universe variables. This restricts the number of possible solutions for the MAX-SAT optimization and the solver even might not find a solution. Further, adding the annotations to the Java source code after the Runtime Inference needs additional time- and memory-consuming steps (parsing the Java sources, parsing the annotation XML file, adding the annotations in the AST, and printing the AST to a source file). The additional work is not advantageous for the performance.

2.4.2. Fixing Types

The UTI Interface offers the concept of fixing and preventing types (see Figure 2.1). An Universe type of an `UtiVariable` can be set by calling the methods `fixType(CUniverse)` and `preventType(CUniverse)`. Using this interface we do not have to insert the Universe

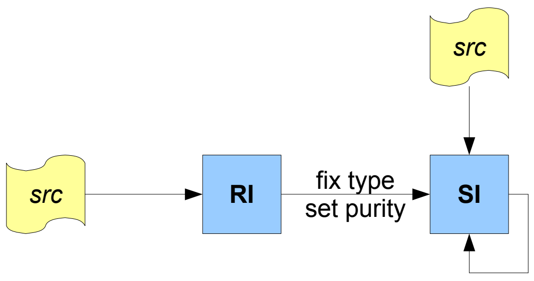


Figure 2.1.: Combination using the fixType interface of UtiVariable.

modifiers in the Java source after the RI step, so that we achieve better performance. The problem of restriction of the domain still stays the same as mentioned in 2.4.1.

2.4.3. Setting Weights

The UTI Interface has a very flexibly weight mechanism that allows to set preferences about annotations. The preferences are respected in the MAX-SAT solver back-end. Using the annotation results from the Runtime Inference and set the appropriate weights for the Static Inference serves as a hint and still leaves a high degree of flexibility for the Static Inference. See Section 2.6 about how the weights are calculated and mapped to preferences.

We think that using these weights gives the user a comprehensive, but intuitive and flexible mechanism to adapt and express preferences about the inference constraint system (see Figure 2.2.

2.4.4. Bad Cast Verification

Using this combination approach we would run a Static Inference first. If the Static Inference introduces type casts, we double-check the correctness of the cast using the Runtime Inference. Actually, we would not use the Runtime Inference itself, but consider the EOG alone. Since the EOG outlines the runtime behavior, that would allow us to verify the casts and eliminate bad casts that fail at runtime.

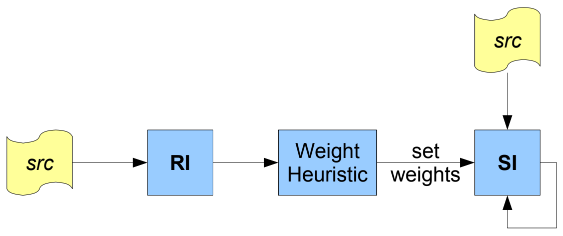


Figure 2.2.: Combination by setting the weights for the Static Inference.

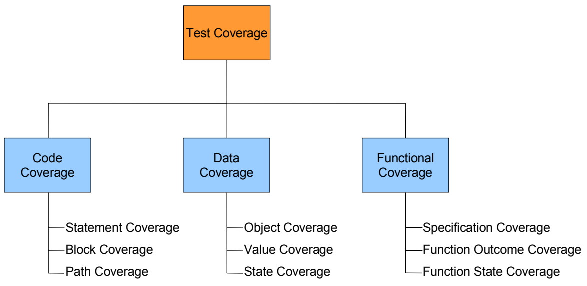


Figure 2.3.: Different models of software test coverage measures.

We did not consider this idea because it is not really an inference combination. The EOG is used to track the runtime behavior of casts and not to infer the Universe types itself. Nevertheless, the elimination of bad casts could be interesting and might be worth for a consideration in future work.

2.5. Determination of Runtime Inference Solution Quality

As we have seen in our assessment of the Runtime Inference, we cannot blindly trust the inferred annotation results. The examples showed that the code coverage and the sequence in which methods are called are essential. We need a way to determine the quality of the Runtime Inference solution. How that can be done for the coverage issue is shown in Section 2.5.1. Some ideas about the method sequences are outlined in Subsection 2.5.2, but we did not follow this topic sufficiently.

There are a lot of different models, patterns, and tools for testing of object-oriented systems ([7] is a good reference for this topic). Figure 2.3 shows an overview about the methods for test coverage measures. Data coverage measures the degree to which the input space for a program is covered. Functional coverage tests whether a program meets the whole specification. Our focus is on the topic of code coverage.

2.5.1. Code Coverage

Code coverage is a measure describing the degree to which the source code of a program has been tested. Code coverage can be considered as an indirect measure of test quality – indirect because we are talking about the degree a test covers our code, or simply, the quality of the tests, not about the quality of the actual product. It helps to identify paths in a program that are not getting tested.

Measures

As always, there are several ways and concepts to measure the code coverage. We will present some widely-used measures and discuss which measure applies best for our requirements.

Statement Coverage: Statement coverage is also known as line coverage. It measures the degree to which individual statements are executed during a test. One advantage of statement coverage is the existence of a lot of tools for this task. It is also quite easy to implement. Measuring statement coverage can be done by instrumenting the bytecode which is a lot simpler than parsing the source code. Another approach is to use a special VM or the JVMTI to get information about executed statements. The disadvantage of statement coverage can be shown in an example:

```
 1  public class UniverseStatementCoverage {
 2      readonly Object x;
 3      peer Object y;
 4      rep Object z;
 5
 6      public void statementCoverageExample(int i) {
 7          if (i > 0)
 8              x = new rep Object();
 9          else
10              x = new peer Object();
11
12          if (i < 0)
13              y = (peer Object) x;
14          else
15              z = (rep Object) x;
16      }
17  }
```

Running the example and performing a call to the method `statementCoverageExample` twice with a value +1 and -1 leads to 100% statement coverage. That may imply we have a good coverage and tested every execution path. Calling `statementCoverageExample` (0) throws a `CastException` even though the coverage was 100%. The **peer** object `x` cannot be cast to **rep** in the assignment on line 15 .

Block Coverage: Instead of reporting individual lines, the block coverage considers each sequence of non-branching statements. For efficiency reasons at execution time it makes more sense to keep track of basic blocks rather than individual lines. Furthermore, block coverage is better in cases as follows:

```
public void bigBlockExample(boolean condition) throws Exception {
    if (condition) {
        System.out.println(”Small␣block␣#1”);
        throw new Exception(”I␣wasn’t␣tested!”);
    } else {
        System.out.println(”Big␣block␣#1”);
        System.out.println(”Big␣block␣#2”);
        System.out.println(”Big␣block␣#3”);
        System.out.println(”Big␣block␣#4”);
        ...
        System.out.println(”Big␣block␣#98”);
    }
}
```

Let us assume that a test invokes `bigBlockExample(false)`. Statement coverage would report a very good coverage of about 98%, even though we have missed an important block. Basic block coverage would consider both blocks equal resulting in 50% coverage. Block coverage is more complex to implement and needs source or bytecode parsing. The problem described in the example method `statementCoverageExample` exists for the block coverage as well. 100% block coverage does not prevent the `CastException`.

Decision Coverage: Decision Coverage, also known as branch coverage, is a measure based on whether decision points (such as if and while statements) evaluate to both `true` and `false` during test execution. The disadvantage is that it does not take into account how the decision points are evaluated. Let us consider the following short example:

```
if (amount > 100 || someCode() == 0) {
   doSomething();
} else {
   doSomethingElse();
}
```

The boolean expression in the `if` statement is considered as one predicate regardless of whether it contains logical AND or OR operators. The decision coverage does not make a statement whether the code for `someCode()` in the OR expression is executed or not.

It is imaginable that the execution of `someCode()` might lead to an exceptional state similar to the one encountered in the statement coverage example.

Path Coverage: Path coverage measures whether each possible path from start (e.g., a method entry) to finish (e.g., a return or throw statement) is executed. Full path coverage is usually impractical or impossible. Any code block with a succession of n decisions can have up to 2^n paths. Loop constructs can result in an infinite number of paths. The advantage of path coverage is the right detection of the code coverage issue like the one that exists in the example method `statementCoverageExample`.

Function/Method Coverage: This measure reports whether each function/method is invoked at least once during the test execution. It is quite simple to implement and an easy way to spot the biggest gaps in the code coverage.

Attribute/Field Coverage: The attribute or field coverage measures whether each attribute of an object is altered at least once.

The question is which of those code coverage measures we should use. Most of the coverage measures and tools are used in the field of software testing. Our focus is a bit different because we are not interested in the correctness of each line of code. Our focus is the coverage in a point of view of the Universe type system. So, we do not need a tool that covers the execution of each line of code. Most of the code coverage measures can be classified as white box testing because the coverage is measured against the internals of classes, not only against the interface or contract of the system. However, a black box testing would be good enough for our purpose.

Also, we do not need a very accurate coverage method. The coverage is only used to get an idea about the quality of the Runtime Inference test case and use this information to set the weights. Full code coverage is reached with the Static Inference in a second step. We are more interested in a good performance of the coverage measure that might reuse as much information about the program execution as we already have gathered from the tracing agent.

As a conclusion of all the mentioned reasons, we decided to use a combination of method and field coverage. We measure whether each function of a class is invoked at least once and if we have a write access to each field.

2.5.2. Method Call Sequences

As mentioned in the introduction to this section, we thought about a solution resolving the *bad method call sequence problem.* Since we want to get more experience in this topic, we did not implement this approach. Nevertheless, we want to outline our ideas briefly.

The fact that we want to distinguish 'good' and 'bad' method call sequences led us to the field of bioinformatics. Sequence comparison algorithms are used in bioinformatics to discover areas of similarity between two sequences of biological data, such as nucleotides in DNA sequences. The goal of these algorithms is not just simply using lexical matching techniques, such as string matching or longest common substring searches, but to also take into account that certain mutations can occur in sequences.

Our approach is to map the bad method call sequence problem to a sequence comparison as it is done for DNA sequences. We build a sequence string by assigning a unique identifier to each method. The sequences of method calls can be extracted from the tracing files. Using the sequence comparisons we think that we can perform a classification between 'good' and 'bad' method call sequences. If a method call sequence has a large distance to a sequence, from which we know that it is good, we say that this is a bad method call sequence. In the same manner we can make statements about sequences with short distances. The closer a specific seqence is to a good sequence, the higher we set the preferences in the Static Inference.

The distances could be measured using one of the following similarity algorithms:

Hamming Distance

The Hamming distance is defined as the number of bits which differ between two binary strings. In other words, it measures the number of bits which need to be changed to turn one string into the other.

Let us consider an example. The strings
`10011011` and
`10001100` have a Hamming distance of 4 bits.

The Hamming distance can only be used for strings of the same length. That is why it does not meet our requirements. Method call sequences can be of different lengths.

Levensthein Distance

The Levensthein distance is an example of an edit distance metric. Edit distance metrics measure the cost of the transformation of one string into another using operations, such as character copy, insertion, deletion, and substitution.

The edit operations for the Levensthein distance are as follows:

- Copy a character from string1 over to string2 (cost 0).
- Delete a character in string1 (cost 1).
- Insert a character in string2 (cost 1).
- Substitute one character for another (cost 1).

The Levensthein algorithm is an application of a bottom-up dynamic programming algorithm.

Smith-Waterman and Monge-Elkan

The Smith-Waterman[31] algorithm is similar to the Levensthein distance, but adds variable cost values for substitution and gaps (deletion or insertion). Monge-Elkan[27] is an affine variant of the Smith-Waterman distance function. Affine means that it assigns a lower cost to a sequence of insertions or deletions. Further, Monge-Elkan is scaled to the interval $[0, 1]$.

Jaro-Winkler

The Jaro-Winkler[21, 33] metric is not based on an edit distance model. It is based on the number and order of the common characters between two strings. While the edit distance metrics are designed to find similarities between DNA sequences, the Jaro-Winkler metric has a different focus. It was designed for linking records in huge databases (e.g., record linking in address databases or linkage of cencus data). Therefore, the Jaro and Jaro-Winkler metrics seem to be intended for short strings like matching of personal and last names.

As an exercise we applied the algorithm and noted the calculations down. The example can be found in Appendix F.

Evaluation and Conclusion

We did some simple evaluation of the discussed distance metrics. The examples are computed using the Java open source library SecondString[10]. We compared the character sequence AAAAAAAA with the sequences AAAABBBB, ABABABAB, and ABABABBA.

metric/string	AAAAAAAA	AAAABBBB	ABABABAB	ABABABBA
Levensthein	0.0	-4.0	-4.0	-4.0
Jaro-Winkler	1.0	0.8	0.7	0.7
Smith-Waterman	16.0	8.0	5.0	4.0
Monge-Elkan	1.0	0.5	0.275	0.225

The Levensthein algorithm only measures the pure edit operations. It does not take into account that ABABABBA is more random than AAAABBBB and, therefore, would probably be a better method sequence. The result with Jaro-Winkler looks similar. It does not reflect the difference between the strings in the last two columns. The distribution of the values for Smith-Waterman and Monge-Elkan seem to meet our requirements best.

The edit distance metrics are dynamic programming algorithms. Consequently, the memory consumption is quite high, especially for long sequences. The Jaro and Jaro-Winkler metric are good for short strings. However, our method call sequences can be quite long.

To perform the classification we need some initial sequences with which we compare our call sequences. It is not obvious how to generate these initial sequences and how to make a semantical statement about ‘good’ and ‘bad’ of the initial sequence. This point is crucial and needs to be investigated furthermore.

2.6. Weight Functions

As introduced in Subsection 2.4.3, we use the Runtime Inference to determine weights and use these weights for the Static Inference. The Static Inference solves an optimization problem by taking the weights into account and returning the solution that maximizes the weights.

In order to set appropriate weights we use heuristics. A heuristics considers the result of the Runtime Inference and calculates some weight contributions. Each heuristic returns a weight contribution for all Universe modifiers **rep**, **peer**, and **readonly**. After all heuristics have calculated their contributions we combine them into a so-called preference and pass it to the Static Inference.

Subsection 2.6.1 and 2.6.2 discusses how preferences are expressed in the Static Inference. The mapping of the heuristic contributions into preferences is outlined in Subsection 2.6.3. We discuss some heuristics in Subsection 2.6.4, 2.6.5, and 2.6.6.

2.6.1. Interpretation of the Weights

The Static Inference tries to annotate a program in such a way that the solution meets the requirements specified with weights. These weights can be set in the inference launch configuration or they are predetermined by the heuristics discussed in the following subsections.

In this subsection we describe how the weights have to be understood and how they are set. The concept of these weights was introduced by Matthias Niklaus[29]. For reasons of comprehensibility we will recall the interpretation of the weights.

What we described as weights so far is grouped in the Static Inference into the following categories:

- Annotation Preference
- Cast Acceptance

An annotation *preference* is specified for each kind of Universe type. In such a way it is possible to express a relation between allowed Universe type modifiers. The preferences are placed in a finite range that goes from minus 100 to plus 100. The default value is zero.

The higher the value, the higher a preference for an annotation with the chosen modifier is. Positive values mean a preference for a modifier. Negative values mean that we do not like that modifier. Let us consider an example using Figure 2.4. We can see the preferences for a Universe type modifier of an instance field, for creation of an instance, parameters of public methods, and parameters of static methods. The weight for modifier of instance fields (see right-hand side of figure) are set to zero for **peer** and **readonly**. The weight for **rep** is set to the value +60. This means that a field of an instance should be annotated with **rep**. If that is not possible, then it does not matter whether a field is annotated as **readonly** or **peer**. Priorities are used to express the importance to fulfill a specific preference compared with other preferences.

The *cast acceptance* is a property that states how many casts are accepted to reach a good solution. Like the other preferences the cast acceptance is a value between 0 and 100, but only positive values are allowed. The higher the value the more casts we accept.

2.6.2. Global and Local Preferences

We extended the preference data structures and introduced a distinction between two types of preferences, called *global preferences* and *local preferences*.

Global Preferences

Global preferences are the same as the existing preferences used by Niklaus. A global preference expresses how a certain kind of member of *any* class should be annotated. This allows us to make statements like:

- Annotate fields with a higher priority than other class members.
- Fields are best to be **rep**.
- Parameters and return types are best to be **readonly**.
- Use as few **peer** annotations as possible.

Global preferences can be characterized with the three properties kind, affiliation, and visibility:

- Kind: Specifies the situation where the annotation occurs
  - Field: The declaration of a field.
  - Local: The declaration of a local variable.
  - Parameter: The declaration of a parameter. The affiliation is related to the method in that case.
  - Return type: The return type. The affiliation is related to the method in that case.
  - Creation: The explicit instantiation of an object. The affiliation is related to the scope where this expression occurs. For the creation **peer** and **rep** modifier are the only modifiers that are allowed.

- Affiliation: States whether the item or the current scope belongs to an instance or to a type.
  - Type: Static fields or items defined in a static context. For all preferences the modifier **rep** is prohibited.
  - Instance: All non-static situations.
- Visibility: States which access modifier is used for the item or the scope.
  - Public
  - Private
  - Protected
  - Default access

Local Preferences

Local preferences are used to express annotation preferences for specific Java class members. The members are identified by a unique name identifier (see Section 4.1.2 on page 60). For instance, we can state how a certain field or method return type should be annotated. In contrast to global properties, the local properties do not respect the affiliation and visibility properties. The local preferences are characterized by the properties

- Kind: Specifies the situation of the local property
  - Field: The declaration of a field.
  - Parameter: The declaration of a parameter.
  - Return type: The return type of a method call.
- Identifier: A name that uniquely identifies the member.

Since we do not use the Runtime Inference to annotate method bodies we only respect the members in the class scope and do not state anything about method body behavior like local variable declaration or object creation. Conceptionally, the kinds “Creation” and “Local” could also be considered in local preferences. But our current implementation does not do that.

Interaction Between Local and Global Preferences

The preferences are looked up when the Static Inference generates the constraints for the inference back-end. For each Java class member that is getting annotated with a Universe modifier, a lookup in the local preferences table is performed. If the Runtime Inference did suggest an annotation we do have an entry in the local preferences and return the corresponding weight. In the other case where we do not have a local preference, we revert back to the global preferences and return the global weight for the corresponing member kind.

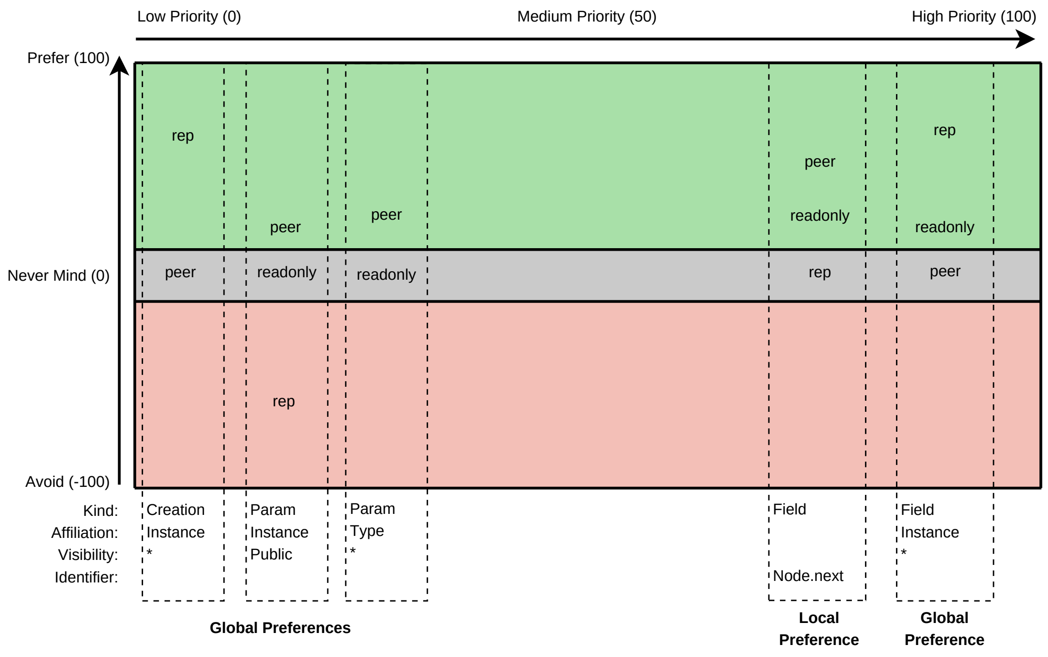


Figure 2.4.: Weights for the Universe annotations. Figure adapted from Niklaus[29]

2.6.3. Setting the Weights for Local Preferences

The Universe type modifiers that are determined by the Runtime Inference and chosen as a suggestion for the Static Inference need appropriate weights. This subsection shows how the weight contributions are combined and mapped to preference weights.

As outlined in the introduction to Section 2.6, each heuristic i in the system returns a *weight contribution* for all Universe modifiers. After all heuristics calculated the weight contribution, we combine the contributions into a local preference.

To describe the weight calculations we use the following notations:
The notation $name\langle v_1, v_2, ..., v_k\rangle$ denotes a tuple *name* with the tuple values v_1 to v_k. Helper functions are denoted with an overline $\overline{h}$.

OM: $OM = \{rep, peer, readonly\}$
the set of Universe modifiers.

u: $u \in OM$
an arbitrary Universe modifier.

$u_{var,k}$: $u_{var,k} \in OM$
The Universe modifier of a variable *var* that is suggested by the Runtime Inference in test run k.

$h_i(u_{var,k})$: $u_{var,k} \rightarrow wc_i\langle wc_{rep}, wc_{peer}, wc_{readonly}\rangle, \qquad wc_u \in [-1, +1]$
the *weight contribution function* of a heuristic i for the variable *var* in a test run k. Returns a tuple (the *weight contribution*) consisting of three elements

$pref_{var,k}\langle \omega_{rep}, \omega_{peer}, \omega_{readonly}\rangle$: $\omega_u \in [-100, +100]$
the preference weight tuple. These weights are assigned to a preference and used as weights in the MAX-SAT solver.

combine: $wc \times wc \times \ldots \rightarrow pref \quad \in [-\omega_{LocalPrefs}, +\omega_{LocalPrefs}]$
Function that combines several weight contributions to a single *pref* tuple.

$\omega_{LocalPrefs}$: $\omega_{LocalPrefs} \in [-100, +100]$
Constant which defines the maximum weight that is set for a combination of weight contributions.

The *combine* function is used to combine and normalize the weight contributions of each heuristic i. It also maps the weight contribution values to a preference. The calculated preference weights are bounded by $\pm\omega_{LocalPrefs}$ in order to set how strong the local preferences should be regarded in relation with the global preferences. The *combine* function is defined as follows:

$$prefs_{var,k} = combine(wc_1, wc_2, ..., wc_n) := \omega_{LocalPrefs} \cdot \frac{\sum_{i=0}^{n} h_i(u_{var,k})}{n}$$

The interpretation of the weight contributions $wc_{var,k} = h_i(u_{k,var})$ is as follows: if a heuristic contributes +1 that means we do like a certain annotation and prefer a preference weight close to $\omega_{LocalPres}$. A value of 0 means that a heuristic does not make a contribution; a value of -1 expresses that we prevent an annotation by setting the preference weight close to $-\omega_{LocalPreference}$.

2.6.4. Coverage Heuristic

As we have seen in Section 2.2.1 the quality of the Runtime Inference directly depends on the code coverage. That is why we want to set the weights based on the value of the code coverage. Good code coverage implies high weight values while bad coverage implies lower weights.

Code coverage is expressed as a percentage. We use a value between 0 and 1. Consequently, a value of 0.7 means that 70% is covered. The meaning of this depends on what form of code coverage has been used, as 70% path coverage is more comprehensive than 70% line coverage. To define what good code quality is, we considered the following empirical evaluations:

The Agitar open quality initiative [17] publishes monthly updated reports about quality of different software projects. Agitar measures the code coverage of some open source projects that is achieved with the unit test cases provided together with the source code. Table 2.1 shows an excerpt of the results. The number of methods and lines in the first and second column give an idea about the size and complexity of the projects. The third column states how many of the classes are tested with JUnit test cases and the last column shows the code coverage achieved by the test cases. Agitar uses a combination of line and decision coverage as the coverage metric. As we see, none of the projects' test cases cover more than 77.8% of the code.

Another evaluation is taken from Binder[7] and refers to [16]. The experiment was conducted on the two widely used utility programs TEX and AWK. Both programs were instru-

Project	Number of Methods	Executable Lines	Classes with Test Points	Coverage
dom4j	2510	6416	27.8%	52.0%
Jakarta-Commons Collections	3539	10429	36.8%	77.8%
jdom	839	3426	21.1%	38.2%
Lucene	1583	7339	33.2%	67.4%
Spring	9880	32125	35.8%	23.0%

Table 2.1.: Coverage report of some widely-used open source projects.

	Block	Decision	p-use	c-use
TEX	85%	72%	53%	48%
AWK	70%	59%	48%	55%

Table 2.2.: Coverage reported by [16] “p-use”= pathways between where a variable is assigned and where it is used in a conditional. “c-use”= pathways between where a variable is assigned and where it is used, but not in a conditional.

mented and the code coverage using the published test suites was calculated. The results are shown in Table 2.2.

Considering these examples we see that, taking an approximation for decision coverage, the best values are around 80%. Binder states that 80% to 85% branch coverage can be achieved. Reaching 100% coverage is not profitable either. A lot of effort is needed to create test cases to reach such perfection.

We define our coverage heuristic as follows:
Let us say that 80% coverage is good coverage, so we want to set high weights. Contrary, if a program inspection reaches 40% coverage, we assume that this program does not have good coverage. We do not want a weight that is the half of the weight of a well covered program with 80% coverage. We assume that 40% coverage is low and want to set a low weight as well. Therefore, our coverage heuristic is not a linear function. Instead we use an exponential function to model this behavior.

$CodeCoverage(var, k)$: the code coverage of a class that contains var in the test run k.

The *coverage contribution function* for the coverage heuristic is defined as follows:

$$h_{coverage}(u_{var,k}) := wc_i\langle\overline{h}(rep, u_{var,k}), \overline{h}(peer, u_{var,k}), \overline{h}(readonly, u_{var,k})\rangle$$

where

$$\overline{h}(u, u_{var,k}) \begin{cases} CodeCoverage(var,k)^3 & \text{if } u = u_{var,k} \\ \dfrac{1}{(CodeCoverage(var,k)+1)^3} & \text{if } u \neq u_{var,k} \quad \wedge \\ & CodeCoverage(var,k)^3 > \frac{1}{(CodeCoverage(var,k)+1)^3} \\ CodeCoverage(var,k)^3 & \text{if } u \neq u_{var,k} \quad \wedge \\ & CodeCoverage(var,k)^3 < \frac{1}{(CodeCoverage(var,k)+1)^3} \end{cases}$$

The functions are chosen such that the coverage weight contributions are mapped into a finite range $[0, 1]$.

The case 2 in the helper function $\overline{h}$ regards the fact that the Runtime Inference can make a wrong suggestion if we have bad code coverage. Therefore, we have to give the Static Inference a certain degree of flexibility by not giving too low weights for the "other modifiers" $OM \setminus u_{var,k}$ that were not suggested by the Runtime Inference.

The case 3 ensures that we never want to assign a higher value than the one suggested by the Runtime Inference. The alternative types should never be treated better than the result from the Runtime Inference. If the alternative value is higher we simply set the weight contribution equal to the one of the suggestion. In that way we say that we give the same preference to all Universe types:

2.6.5. Parameter Heuristic

In the Universe type system methods are executed in the context that contains the receiver object. Method parameters and return types are relative to the context of the receiver. Consequently, method parameters and return types are best to be **readonly** or **peer**. To get deep universe structures, we prefer **readonly** to **peer**.

These properties are expressed in the parameter heuristic. The parameter heuristics assigns a weight factor to each ownership type of a parameter:

- the weight contribution of **readonly** parameters is set to 1.
- the weight contribution of **peer** parameters is set to 0.8.
- the weight contribution of **rep** parameters is set to 0.5.

$$h_{param}(u_{var,k}) := wc_{param}\langle \overline{h}(rep)), \overline{h}(peer), \overline{h}(readonly)\rangle$$

where

$$\overline{h}(u) \begin{cases} 1.0 & \text{if } u = readonly \\ 0.8 & \text{if } u = peer \\ 0.5 & \text{if } u = rep \end{cases}$$

2.6.6. Field Heuristic

Instance fields are best to be **rep**. That is why we set the weight contribution to 1.0 if the suggested ownership modifier for a field is **rep**. If the Runtime Inference inferred a **peer** or **readonly** solution we decrease the weight contribution to 0.8.

A possible extension would be to respect the visibility modifiers in this heuristic as well. If a field is `private`, it is more likely to be **rep**. A `public` field is more likely to be **peer** or **readonly**. Currently this is not implemented in this field heuristic.

Chapter 3.

Eclipse User Guide

This chapter is a description of the implemented Eclipse plug-ins from a user's point of view. The Eclipse User Guide should be relatively independent from the rest of the report, so that it could easily be used as a documentation for plug-in users. Implementation or architectural design details are not described in this chapter, but can be found in Chapters 2 and 4.

3.1. Installation and Configuration

To run the plug-ins Eclipse 3.2 is required. We developed and tested our implementation on Linux (Ubuntu with kernel 2.6.17). We have also successfully run the project under Windows XP. Mac OS X is not supported because the PBS solver is only available as a binary for Windows, Linux, and Solaris.
The Universe tool plug-ins are packed in three different plug-in features:

- **JML Tools** The JML Tools (JML checker, JML compiler)
- **Universe Type Inference** All the Universe type inference plug-ins
- **Universe Visualizer** The graphical visualization of the Universe structure.

3.1.1. Plug-in Installation

Online updates are available on `http://sct.inf.ethz.ch/research/universes/tools/eclipse/`.

Define a site bookmark in Eclipse's Update Manager view as follows:

1. Choose **Software Updates ▶ Find and Install** from the **Help** menu.
2. Select **Search for new features to install** and press **Next**.
3. Press **New Remote Site...**
4. Enter "Eclipse Universe Tools" (or whatever you like) as the name, and the Site URL mentioned above.
5. Press **Finish**
6. Check the boxes next to the features and versions you would like to install and press **Next**.
7. Accept the license agreement and press **Next**.

8. Press **Finish**.
9. Eclipse may ask if you want to restart. We recommend that you do.

3.1.2. Solver Installation

Due to the curious license of the PBS solver we are not allowed to deliver the solver binary together with our plug-ins. Therefore, you have to download and install the solver yourself.

1. Go to `http://www.eecs.umich.edu/~faloul/Tools/pbs/` and download the PBS solver
2. Extract the archive somewhere on your machine.
3. Start Eclipse and choose **Preferences...** ▶ **Universe Type Inference** ▶ **Static Inference** from the **Window** menu.
4. Click on the **Browse...** button and select the binary of the PBS solver (see Figure 3.6).
5. Press **Apply** to save the preferences.

3.1.3. Logging Settings

The Runtime and the Static Inference produce a bunch of log messages during the execution of the algorithms. These messages are classified in different levels and shown in the standard Eclipse "Error Log" view. To keep the control over all logging message, the logging behaviour is configurable as follows:

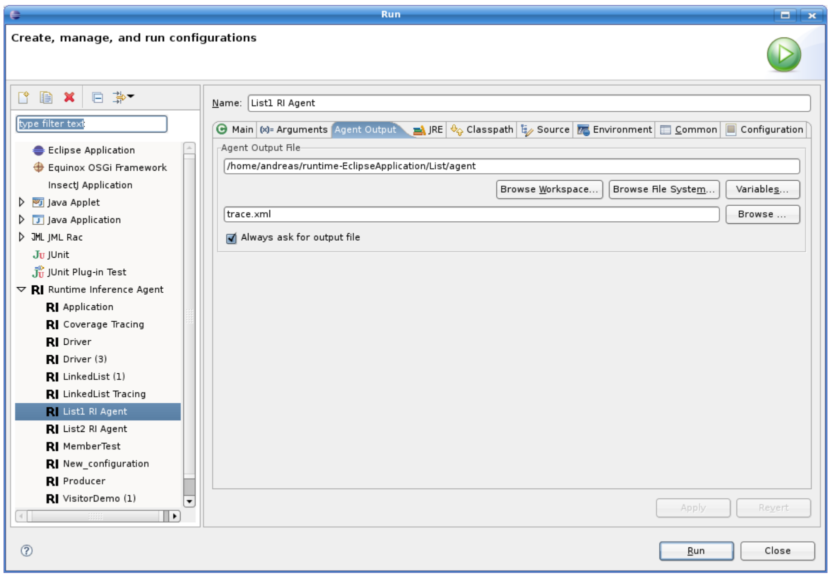
1. Open the workspace preferences by clicking the menu **Window** and selecting **Preferences...** ▶ **Universe Type Inference** ▶ **Logging**.
2. Check the items for the different log levels you want to have logged.

debug	Shows all debugging log events
info	Shows the info status log events
warning	Shows warnings
error	Shows log events for errors
fatal	Shows fatal error log events

3. Click on **Apply** to save the settings persistently. The settings are valid for all projects in your current workspace.

3.2. Runtime Inference Plug-in

The Runtime Inference plug-in automatically infers Universe type annotations from executable Java programs. This is done through a runtime observation and without static code analysis of the Java program. The inference process is subdivided into two parts. The first step is to monitor the program execution with a special tracing agent that is attached to the Java virtual machine. The second is the type inferer itself which uses the tracing information for the type deduction.



Name: List1 RI Agent

Agent Output File

/home/andreas/runtime-EclipseApplication/List/agent

trace.xml

☑ Always ask for output file

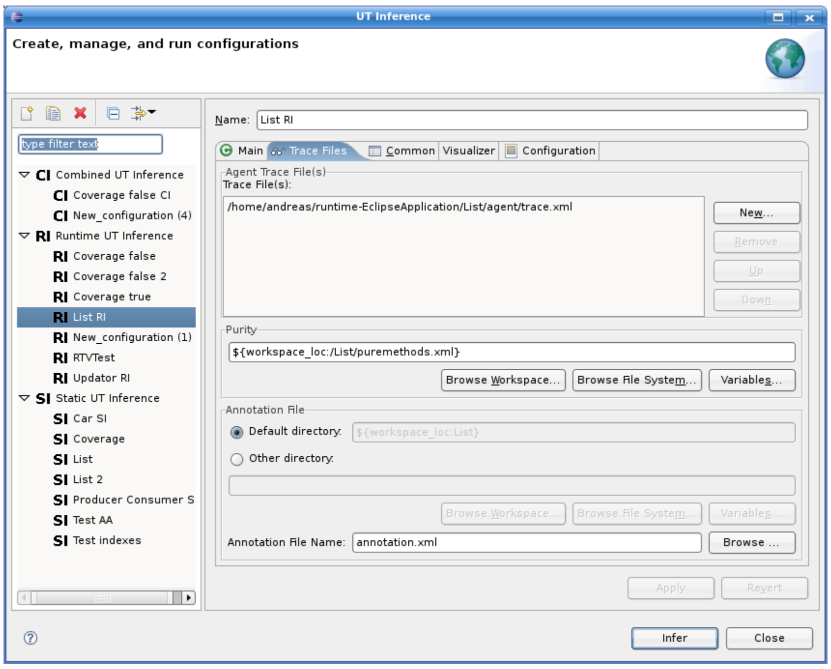
Figure 3.1.: Launch configuration with the Agent Output tab.

3.2.1. Tracing Agent

The tracing agent can be started and configured in the new launch configuration type "Runtime Inference Agent". Running such a launch configuration creates a new virtual machine process with the attached tracing agent and monitors the program execution. As every executable Java application, at least one of the inspected classes has to have a main method.
To create a new Tracing Agent launch configuration right-click on the desired Java source file and select **Run As ▶ Runtime Inference Agent**. Alternatively, if you want to specify arguments for the program or the destination of the tracing file you have to open the Launch Configuration dialog by selecting **Run...** in the Run drop-down menu. Create a launch configuration under the category "Runtime Inference Agent". A launch configuration allows you to configure how a program is launched, including its arguments, classpath, and other options. The Java specific tabs are identical to the ones under the launch configuration for normal Java applications. Additionally, we have the "Agent Output" tab that is used to specify the destination of the tracing file (see Figure 3.1). The user can choose whether or not he wants to be asked for the name of the output file before each run of the tracing agent. The tab "Configuration" shows the command that has to be executed in order to run the agent on the command line.

The input and output of the tracing agent is processed in the "Console" view. You get notified about the termination of the tracing agent with the status message "tracing done." in the "Console" output. The project directory is also updated as soon as the tracing agent is terminated in order to see the the tracing file in your project. If the inspected program needs input from the command line you can enter these directly in the "Console" view.

Create many different traces with different inputs to get good code coverage and consequently get a better inference result.



Name: List RI

Trace File(s): /home/andreas/runtime-EclipseApplication/List/agent/trace.xml

Purity: ${workspace_loc:/List/puremethods.xml}

[✓ Default directory: ${workspace_loc:List}]

Annotation File Name: annotation.xml

Figure 3.2.: Inference configuration for the Runtime Inference.

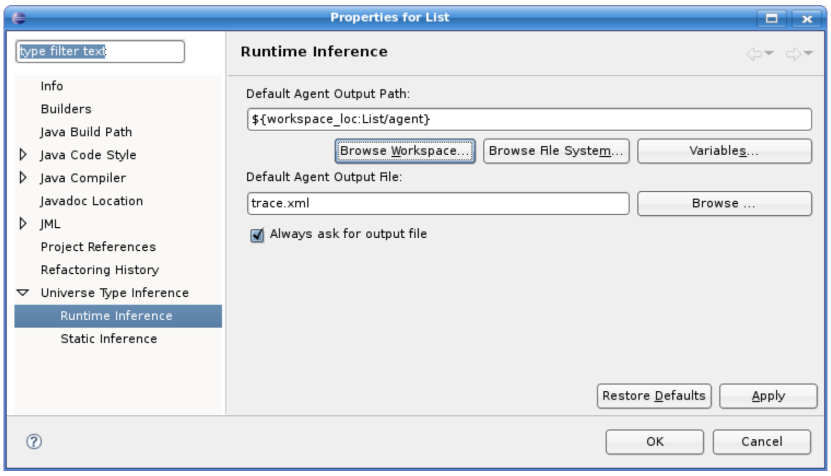
3.2.2. Type Inferer

Similar to the launch configurations for launching programs we have inference configurations for launching a Universe Type inference for a program. Open the UT Inference dialog by selecting the menu **Inference ▶ Run Inference Configuration** or by clicking the icon in the toolbar. Create an inference configuration for the inference type "Runtime UT Inference" (see Figure 3.2).

The "Main" tab is similar to the main tab for Java applications and is used to specify the project. The tab "Inference Files" (see Figure 3.2) allows to specify input and output files of the inference. Add all trace files to the list "Agent Trace File(s)". Optionally, you can select the purity file with a few purity annotations in the "Purity" text field. The output directory and the annotation XML file is specified in "Annotation File". On the "Configuration Tab" there is an **Export XML...** button that saves the current configuration in a configuration XML to be used when you run the type inferer on the command line.

3.2.3. Project Properties

The Runtime Inference plug-in has some project properties which can be managed in an own property page (see Figure 3.3). Precisely, these are the settings for the output path and the file name of the annotation XML output file. To open the property dialog right-click on the project in the Package Explorer and select **Properties ▶ Runtime Inference**. The properties set in this property page are valid as long as they are not overwritten in a specific Runtime Inference launch configuration as we have seen above in 3.2.1. The default values for these property settings are stored in the Runtime Inference Resources plug-in within the file `runtimeinference.default.properties`. Changing these default values does not require



Runtime Inference

Default Agent Output Path:

${workspace_loc:List/agent}

Default Agent Output File:

trace.xml

☑ Always ask for output file

Figure 3.3.: Runtime Inference Project Properties.

to recompile the plug-in project. The path value can contain an absolute path or use path variables as shown in the screenshot.

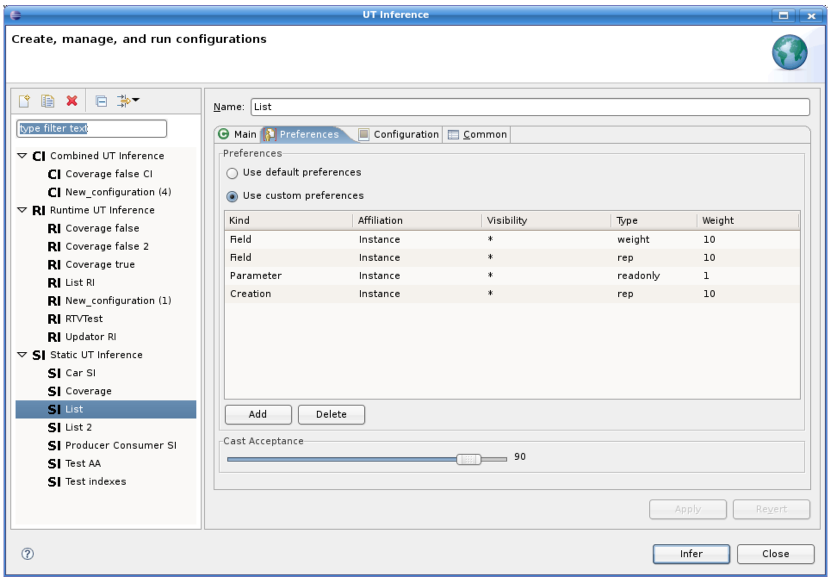
3.3. Static Inference Plug-in

The Static Inference plug-in automatically infers Universe type annotations from Java sources. This is done through a static code analysis of the Java source code. The result of the Static Inference is shown in an own view and can be edited and refined by the user.

3.3.1. Type Inferer

Similar to the inference configuration for the Runtime Inference (described in 3.2.2) there is an inference configuration for the Static Inference. Open the UT Inference dialog and create an inference configuration for the inference type “Static UT Inference” (see Figure 3.4).

The “Main” tab serves as a selection for the actual project and is pretty self-explaining. More interesting is the tab “Preferences”. It allows to specify the preferences for the Static Inference. We focus here on the handling and setting of the preferences. See Section 2.6 for more detailed explanations of the concept of preferences and weighting functionality.
You can use the default preferences or set customized preferences that are better suited for your current project. Notice that no default settings are used when you select the radio button “Use custom preferences”. To add a new preference click on the button “Add” and directly edit the new entry in the table. To delete a preference select the candidate in the table and click the button “Delete”. The value for the weight must be in the range -100 and +100. The cast acceptance can be selected with the slider and goes from 0 to 100. A cast acceptance of 100 means that you allow a lot of casts, a value of 0 means that you do not want casts necessarily. In the “Options” tab you can select how pre-annotated sources are handled. There are the two options “Fix Types” and “Set Weights”. The mode “Fix Types” is more restrictive and fixes the types. The solver might not find a inference solution because of too strong restrictions in the constraints. The mode “Set Weights” is less restrictive. It simply



- CI Combined UT Inference
  - CI Coverage false CI
  - CI New_configuration (4)
- RI Runtime UT Inference
  - RI Coverage false
  - RI Coverage false 2
  - RI Coverage true
  - RI List RI
  - RI New_configuration (1)
  - RI RTVTest
  - RI Updator RI
- SI Static UT Inference
  - SI Car SI
  - SI Coverage
  - SI List
  - SI List 2
  - SI Producer Consumer SI
  - SI Test AA
  - SI Test indexes

Name: List

Preferences

Use default preferences [✓ Use custom preferences]

Kind	Affiliation	Visibility	Type	Weight
Field	Instance	*	weight	10
Field	Instance	*	rep	10
Parameter	Instance	*	readonly	1
Creation	Instance	*	rep	10

Cast Acceptance 90

Figure 3.4.: The Preferences tab of a Static Inference configuration.

sets some constant weights as local preferences and lets the solver find an optimal solution with maximal weights. Changes of pre-annotations to other Universe modifier are possible.

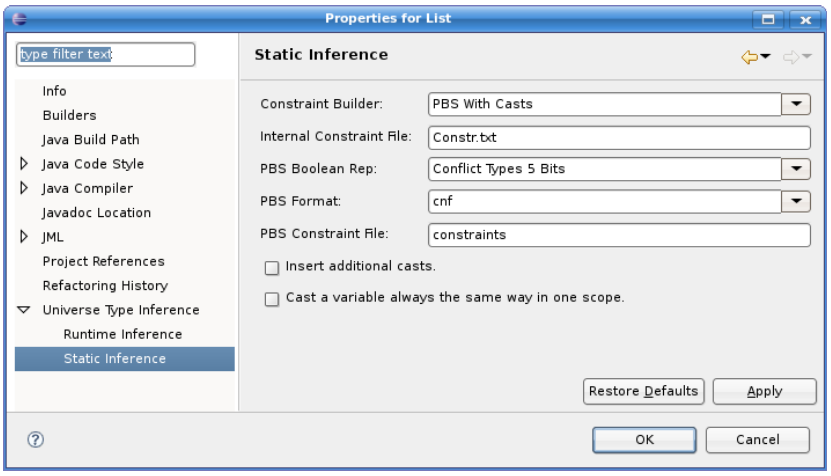
3.3.2. Project Properties

Like in the Runtime Inference plug-in a project in the Static Inference plug-in has also properties which can be managed in an own property page. Open the project properties dialog and click on “Static Inference” to open the Static Inference property page (see Figure 3.5).

The property page allows to set some settings that are relevant for the SAT solver backend. “Internal Constraint File”, “PBS Format”, and “PBS Constraint File” are internal settings that describe how the constraints are passed to the Pseudo-Boolean Solver (PBS). Normally, the default settings are fine and you do not have to touch them. The default settings can be changed in the Static Inference Resources plug-in within the file `staticinference.default.properties` without recompiling the plug-ins. More interesting are the other settings. “Constraint Builder” enables to select whether a constraint builder that accepts casts or not should be used. With the setting “PBS Boolean Rep” the user can choose the type system and its SAT encoding that is used for the inference. Notice that the different type systems lead to different inference results. Please consult the report done by [29] to read more about the type settings and encodings.

3.3.3. Workspace Preferences

In contrast to the project properties the workspace preferences are not valid for only one project, but are persistent for all projects in the workspace. In the case of the Static Inference these are the path and arguments for the PBS solver. Open the workspace preferences by selecting the menu **Window ▶ Preferences... ▶ Universe Type Inference ▶ Static Inference** (see Figure 3.6). The setting “SAT Solver” is the path to the SAT solver (in our



Static Inference

Constraint Builder: PBS With Casts
Internal Constraint File: Constr.txt
PBS Boolean Rep: Conflict Types 5 Bits
PBS Format: cnf
PBS Constraint File: constraints
☐ Insert additional casts.
☐ Cast a variable always the same way in one scope.

Figure 3.5.: Static Inference Project Properties.

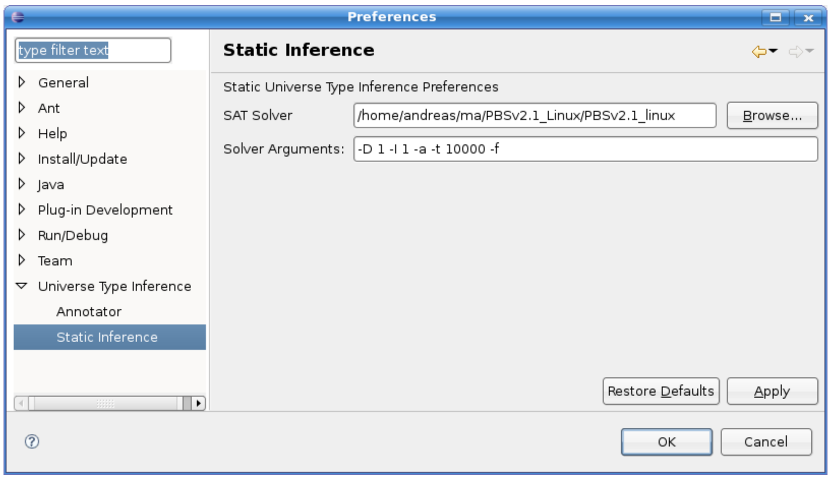
case the PBS solver `PBSv2.1_linux`). “Solver Arguments” are the command line arguments that are passed to the SAT solver during the process creation. See the documentation on [3] about the PBS solver.

3.3.4. Annotation View

After launching a Static Inference launch configuration the result of the inference is shown in the Universe Type Annotation View (see Figure 3.7). The Universe Type Annotation View is something similar to the Java Outline View, except that it does not show all Java members, but only the ones that are relevant for the Universe type system needs. Precisely speaking, it shows types, field declarations, method return values, object creations, local variable declarations, and casts. There is a special icon associated with each kind of annotation (see table below).

Icon	Description
Ⓣ	A **type** containing Universe type annotations
Ⓕ	A **field** declaration
Ⓘ	An **initializer** for a field or array
Ⓜ	A **method** containing Universe type annotations
Ⓟ	A method **parameter**
Ⓛ	A **local** variable declaration
Ⓝ	A **new** object instantiation
Ⓒ	A type **cast**
Ⓡ	A method **return**

A double-click on an item in the Annotation View opens the corresponding Java source in the editor and jumps to the defining position. Also, if the cursor in the source editor is set to a position that has a counterpart in the Annotation View, the item in the AnnotationView is highlighted.



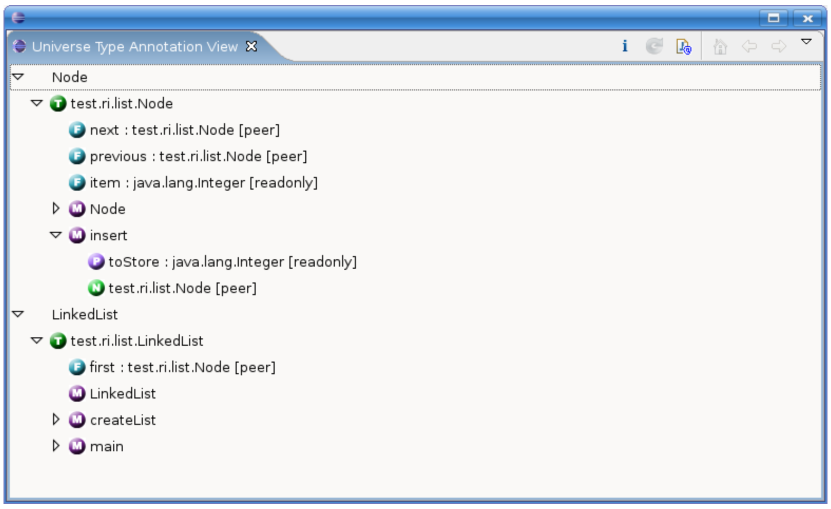
Static Inference

Static Universe Type Inference Preferences

SAT Solver /home/andreas/ma/PBSv2.1_Linux/PBSv2.1_linux

Solver Arguments: -D 1 -I 1 -a -t 10000 -f

Figure 3.6.: Workspace preferences page to set the path and arguments for the SAT backend.



```
Node
  test.ri.list.Node
    next : test.ri.list.Node [peer]
    previous : test.ri.list.Node [peer]
    item : java.lang.Integer [readonly]
    Node
    insert
      toStore : java.lang.Integer [readonly]
      test.ri.list.Node [peer]
LinkedList
  test.ri.list.LinkedList
    first : test.ri.list.Node [peer]
    LinkedList
    createList
    main
```

Figure 3.7.: The Universe Type Annotation View.

Infer New Solution

The Annotation View allows to change some Universe annotations. Right-click on an item in the tree and select the menu to set or prevent a certain type. The purity of methods can be edited as well. After setting or preventing some types, the Annotation View is dirty and the "Infer" button in the toolbar gets enabled. Clicking on the infer button infers a new solution and performs all changes for implicit dependencies. If the inferer does not find a solution it undoes all fix and prevent actions that were done since the last inference.

Notice that the Java source is parsed only once. The abstract syntax tree and constraints are kept in memory resulting in better performance. If you want the sources to be parsed again, open the view menu (arrow in the top-right corner of the view) and select the menu **Parse Sources**.

Insert Annotation Into Source Code

If you are satisfied by the current inference result you can insert the annotations directly into the source code by clicking on the icon in the toolbar. The style of the annotation can be changed in the workspace preferences (see Section 3.4 for details).

Export Annotations

The current solution shown in the Annotation View can be exported into an annotation XML. Click on the icon in the toolbar. A file dialog opens and lets you choose where to store the annotation XML.

Import Annotations

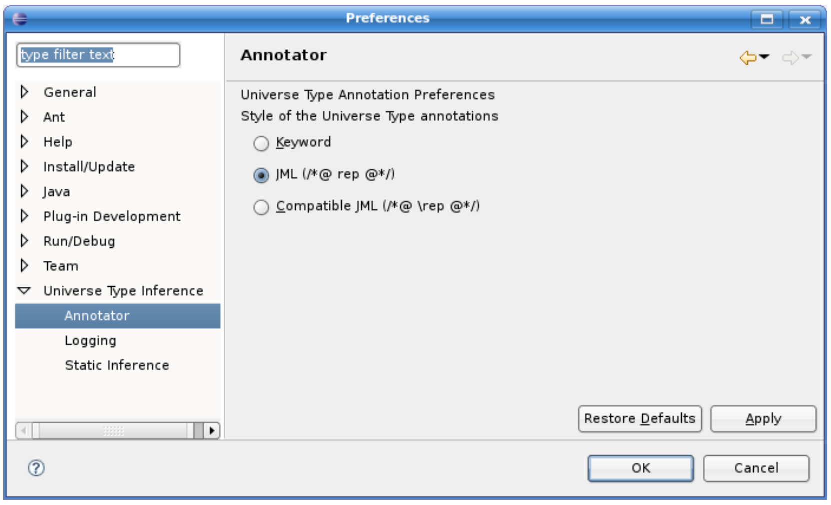
It is possible to import annotations from an XML file and fix the types based on the settings in the XML file. Open the drop-down menu in the top-right corner of the view and select the menu **Fix Types...**. A file selection dialog opens and lets you choose the annotation XML with the pre-annotations. Notice that the import does not change anything in the structure of the types. Only the Universe type fixes of members in the Annotation View that can be matched with an entry in the XML file are performed.

3.4. Annotator Plug-in

The Annotator plug-in is used to insert Universe modifiers into the Java source code. To run the annotator you have to perform the steps described below.

3.4.1. Annotating Java Sources

Right click on a Java file in the Package Explorer or Navigator and select the menu **Annotator ▶ Run Annotator...**. Choose the XML annotation file with the corresponding annotation information in the openend file selection dialog. After clicking OK the annotations are inserted. If you want to insert the annotations into more than one Java file, just select several files or run the annotation action on the whole package by selecting the package



Annotator

Universe Type Annotation Preferences
Style of the Universe Type annotations
( ) Keyword
(●) JML (/*@ rep @*/)
( ) Compatible JML (/*@ \rep @*/)

Figure 3.8.: The workspace preferences for the Annotator styles.

in the Package Explorer.

There are several styles for the annotations:

Style	**Description**	**Example**
Keyword	Use Universe modifiers like a normal modifier keyword. Not compatible with standard Java compiler.	`private rep Obj myObj;`
JML	Use JML comments. Compatible with standard Java compiler.	`private /*@ rep @*/ Obj myObj;`
Compatible JML	Use escaped JML comments. Compatible with standard Java compiler. Ignored if JML is used without Universe support	`private /*@ \rep @*/ Obj myObj;`

The style of the annotations can be selected in the workspace preferences. Select menu **Window** ▶ **Preferences...** ▶ **Universe Type Inference** ▶ **Annotator** (see Figure 3.8).

If you have a source file with unsaved modifications openend in an editor, the Annotator asks you whether you want to save the modifications before the annotation step. After annotating the affected sources are opened in the editor. If you do not like the inserted annotations you can simply discard them by undoing the operation. Click on **Edit** menu and select **Undo Typing**.

Notice that the Annotator plug-in has to parse the source in order to insert the Universe modifiers at appropriate positions. Hence, the original code formattings (whitespaces, indents, linespaces) get lost.

3.4.2. Editing Sources

We use the normal Eclipse JDT editor to edit Java files with Universe annotations. Thus, the whole JDT functionalities are available. We extended the JDT editor with code templates for the Universe annotations. If you want to insert a Universe keyword, you only have to type the Universe modifier keyword (just a few letters are enough) and type **Ctrl + space**. That inserts the modifier in the JML style without finger acrobatics. Notice that Eclipse reformats code templates per default. Therefore, a whitespace is inserted between the * and @ which is not JML conform. The reformatting can be switched off in **Window ▶ Preferences ▶ Java ▶ Editor ▶ Templates**. Unselect the checkbox "Use code formatter" at the bottom of the dialog. Unfortunately, Eclipse applies this setting for all templates.

3.5. Combined Inference Plug-in

The Combined Inference is a combination of both approaches, the Runtime Inference and the Static Inference. In a first step a Runtime Inference is run and its results are used as an input to the Static Inference that is run in a second step.

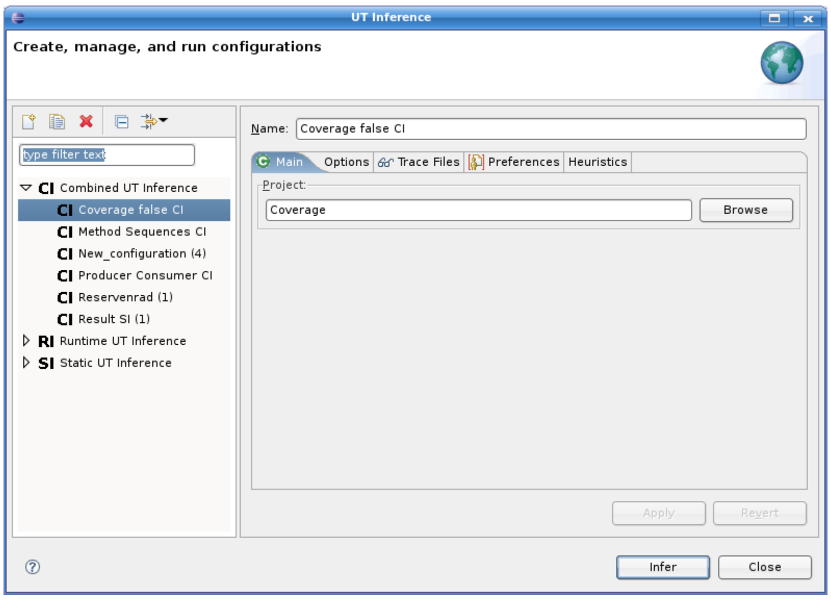
To launch a Combined Inference open the UT Inference dialog and create a new inference configuration of the type "Combined UT Inference" (see Figure 3.9). The tabs for this inference type are the same as for the Runtime Inference (see Section 3.2.2) and Static Inference (see Section 3.3.1) and have to be configured in the same way. Notice that the preferences for the Static Inference, either the default preferences or the custom preferences, are used as global preferences. Local preferences for the annotation of specific fields and parameter types are determined based on the result of the Runtime Inference. The result of the predetermined local preferences are shown in an own dialog (see Figure 3.10). The user can adapt the weights if they do not fit to their needs.

The tab "Heuristics" lists all heuristics that are applied to determine the local preferences for the Static Inference (see Figure 3.11). Selecting a heuristic shows a short description about the goal of the heuristic. With the slider at the bottom you can set the maximum value for a weight that can be set by the heuristics. In the "Options" tab you can additionally select wether you want to run the Runtime Inference or use an annotation XML file that has already been written in a previous inference run. The option how pre-annotations are handled is identical to the one in the Static Inference.

3.6. JML Tools

Paolo Bazzi integrated the JML tools into Eclipse. The JML checker is available in the context menu of each Java source code file. By right-clicking a Java file and choosing the pop-up menu **JML Tools ▶ JML Checker** the JML checker is executed. Errors which are found by the checker are shown in the "JML Error View".

The JML compiler is integrated in a very similar way. Open the pop-up menu of a source file and select **JML Tools ▶ JML Compiler**. Compiler errors are shown in the "JML Error View" and the Eclipse "Problems" view. There is a special launch configuration to run JML compiled classes with runtime assertion checks. Open the launch dialog by selecting menu



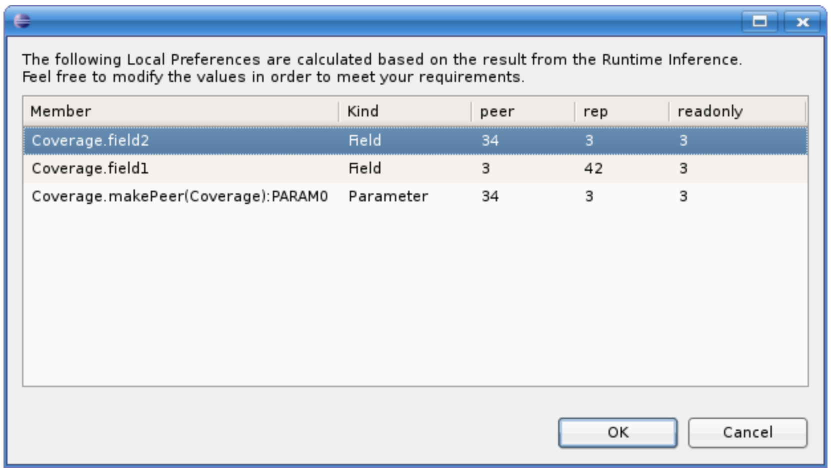
Create, manage, and run configurations

- CI Combined UT Inference
  - CI Coverage false CI
  - CI Method Sequences CI
  - CI New_configuration (4)
  - CI Producer Consumer CI
  - CI Reservenrad (1)
  - CI Result SI (1)
- RI Runtime UT Inference
- SI Static UT Inference

Name: Coverage false CI

Project: Coverage

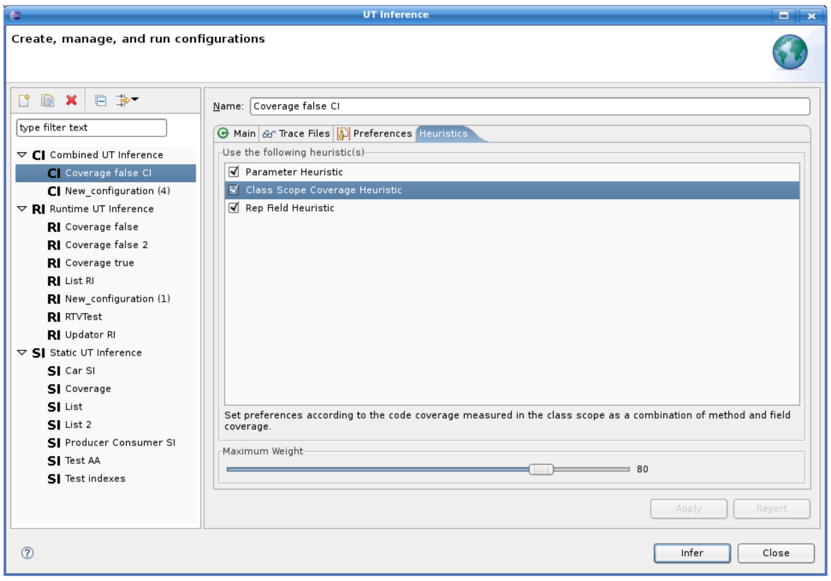
Figure 3.9.: The inference configuration for the Combined Inference approach.



The following Local Preferences are calculated based on the result from the Runtime Inference.
Feel free to modify the values in order to meet your requirements.

Member	Kind	peer	rep	readonly
Coverage.field2	Field	34	3	3
Coverage.field1	Field	3	42	3
Coverage.makePeer(Coverage):PARAM0	Parameter	34	3	3

Figure 3.10.: The weight dialog for the local preferences.



UT Inference

Create, manage, and run configurations

type filter text

- CI Combined UT Inference
  - CI Coverage false CI
  - CI New_configuration (4)
- RI Runtime UT Inference
  - RI Coverage false
  - RI Coverage false 2
  - RI Coverage true
  - RI List RI
  - RI New_configuration (1)
  - RI RTVTest
  - RI Updator RI
- SI Static UT Inference
  - SI Car SI
  - SI Coverage
  - SI List
  - SI List 2
  - SI Producer Consumer SI
  - SI Test AA
  - SI Test indexes

Name: Coverage false CI

Main | Trace Files | Preferences | Heuristics

Use the following heuristic(s)

- ✓ Parameter Heuristic
- ✓ Class Scope Coverage Heuristic
- ✓ Rep Field Heuristic

Set preferences according to the code coverage measured in the class scope as a combination of method and field coverage.

Maximum Weight: 80

Apply | Revert

Infer | Close

Figure 3.11.: The configuration tab for the applied heuristics.

Run ▶ Run... and creating a RAC launch configuration.

Please refer to [5] for more detailed instructions about the usage of the JML tools.

3.7. Inference Visualizer

The Inference Visualizer is a plug-in that graphically visualizes the Universe structure. There are several motivations for the Visualization. The visualization of the Universe structure improves the usability of the Universe type system tools significantly. Obviously, a graphical visualization is more intuitive than reading and manually interpreting annotated source code files. Another goal of the Visualizer is, that it enables to step through the Runtime Inference algorithm and observe the EOG. Future work might extend the Visualizer to interact with the annotation model and the inferer.

3.7.1. Start the Visualization

Runtime Inference

To start the Inference Visualizer you have to open the tab “Visualizer” in the Runtime Inference configuration dialog and select the checkbox “Show Universe Visualization”. If you check the box “Show class fields and methods” the objects are visualized similar to UML class diagrams (see screenshot in Figure 3.12).

If you start the inference, a new editor is opened automatically. Either you click on the play button to go directly through the different steps of the algorithm, or you click on the next step icon which only displays one algorithm step and pauses. The status line shows a short message about the action that was executed by the algorithm at last.

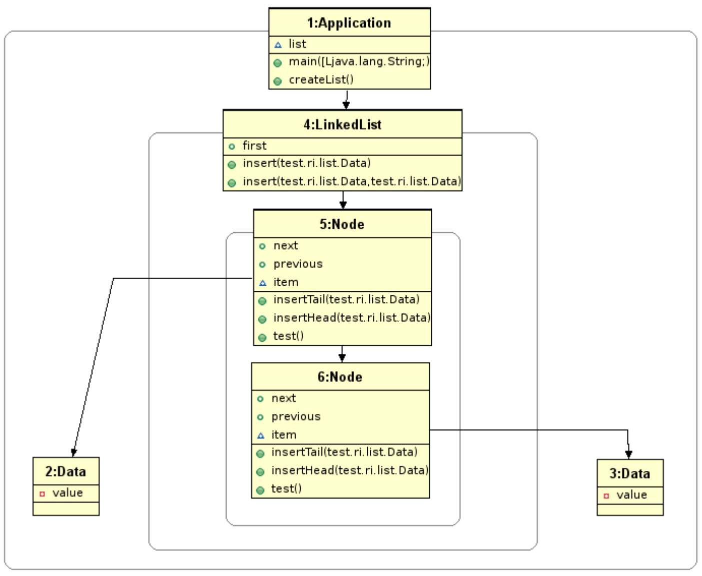

1:Application
list
main([Ljava.lang.String;)
createList()
4:LinkedList
first
insert(test.ri.list.Data)
insert(test.ri.list.Data,test.ri.list.Data)
5:Node
next
previous
item
insertTail(test.ri.list.Data)
insertHead(test.ri.list.Data)
test()
6:Node
next
previous
item
insertTail(test.ri.list.Data)
insertHead(test.ri.list.Data)
test()
2:Data
value
3:Data
value

Figure 3.12.: Screenshot of a Universe structure visualization.

Static Inference

Start a Static Inference and open the "Universe Type Annotation View". Click on the icon in the view's action toolbar. This opens the Visualizer Editor and shows the visualization of the current Universe structure. The current implementation only visualizes the heap structure, that means only field references are shown.

3.7.2. Properties View

If you click on a class object or a Universe in the editor, the properties of the selected object are shown in the properties view. It might be that the property view is not shown in your current Eclipse perspective. In this case, open the property view by selecting **Window ▶ Show View**.

3.7.3. Layout

As a default setting the Inference Visualizer uses an automatic layout that tries to find the best positioning of class instances and Universe bubbles. If you are not satisfied with the automatic layout, you can click on to change to manual layout mode. In the manual layout mode you can move and scale class and Universe nodes.

The references between nodes can be arranged by selecting a reference edge and grab the small icon anchor that appears in the middle of the edge. In this way you can create bendpoints through which the reference edges are routed. To remove bendpoints just drag a connection back to a straight line and the bench points gets removed automatically.

It is also possible to remove reference edges for visibility reasons. Selecting a reference edge and pressing "Delete" removes an edge from the visualizer view. All actions can be undone with the undo and redo actions (see screenshot of the toolbar in Figure 3.13).

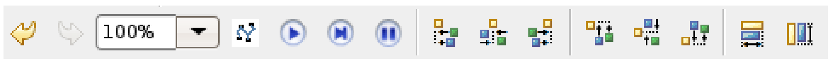


Figure 3.13.: The toolbar of the Inference Visualizer with the actions for undo, redo, the zoom, toggle between automatic and manual layout, the player control, and the alignment.

3.7.4. Zooming

The editor content can be scaled using the zoomer. To zoom in or out you have to select the zoom percentage value in the zoom box that you can find in the toolbar. To keep the overview the outline view shows the big picture with the currently selected excerpt of the editor.

3.7.5. Alignment

Once you changed the layout to the manual mode, you can align class boxes as follows: press the Ctrl key and select several boxes with the mouse. Now you can select an align action to align the selected boxes on the left, right, top, bottom side, or in the middle (see Figure 3.13). The last two icons are the actions to adjust the box size. The width and height of the selected boxes are matched with the size of the primary selection.

Chapter 4.

Implementation

This chapter describes the implementation of our project as Eclipse plug-ins. The introduction into the Eclipse world can be quite painful. But after getting used to all the terms and concepts, we really began to like the Eclipse environment. One can see the influence of the "Gang of Four" members resulting in the clean architecture and the application of design patterns. As references we can recommend the books [8] and [14].

We will start with the overview of the whole plug-in collection and go step-by-step through each plug-in itself.

4.1. Overview

4.1.1. Package Structure

The Eclipse integration of the Universe Type Inferer is distributed over a collection of plug-ins. The motivation is to keep the degree of modularization and reuse as high as possible. We introduced some naming conventions for package names and identifiers of plug-ins or extension points to keep a consistent structure.
The whole integration is implemented as a part of the package `ch.ethz.inf.sct.inference`. Classes related to static inference are in a package called `si`, classes related to runtime inference in a package called `ri`. Notice that it is not possible in Java to use a keyword in a package name. Therefore, it was not possible to chopse the package name `static`. That is why the abbreviations `si` and `ri` are used. Each plug-in resides in a project directory of its own. The Eclipse identifier of a plug-in is identical to the package name.

Eclipse extensions are identified by *extension identifiers*. All extension identifiers are settled at the level of the plug-ins because an extension is valid in the whole scope of the plug-in respectively and not only for a subset of a plug-in.

To keep the system structured we use the following naming conventions for the extension identifiers and package names:

Convention:	plugin-name.extension-name.uniqueExtensionIdentifier
Example:	ch.ethz.inf.sct.inference.common.launchConfigurationType.si

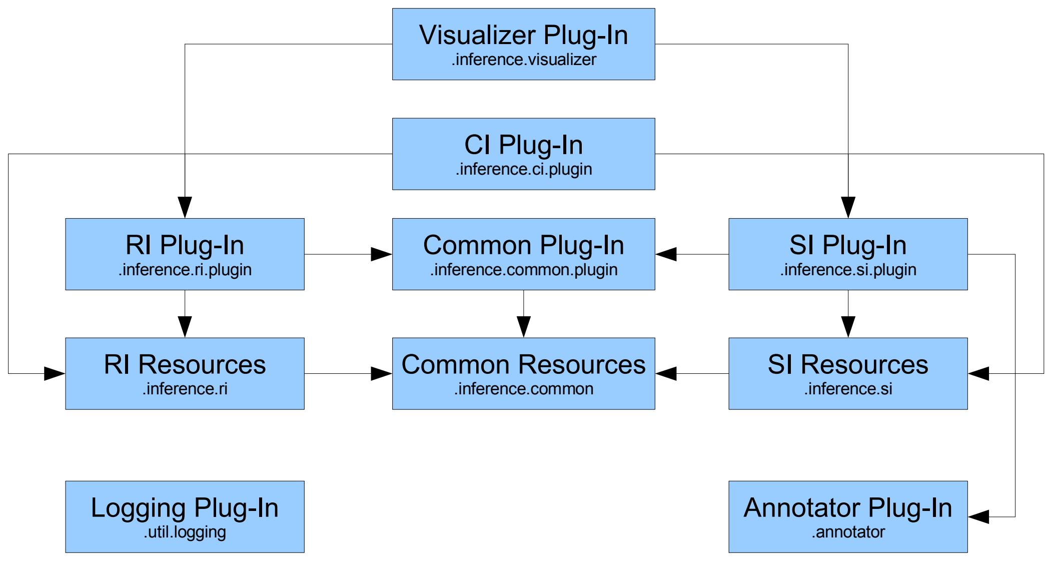


Figure 4.1.: The dependencies between the different plug-ins.

Dependencies between the plug-ins are illustrated in Figure 4.1 and organized as follows:

Plug-in Identifier	Description
inference.si	Recources of the Static Inference tool
inference.ri	Resources of the Runtime Inference tool
inference.si.plugin	Eclipse plug-in code for SI
inference.ri.plugin	Eclipse plug-in code for RI
inference.common	Common resources shared by both resource plug-ins
inference.common.plugin	Common code shared by both plug-ins
inference.ci.plugin	Eclipse plug-in code for the combined inference
inference.visualizer	The graphical Universe visualizer
annotator	Eclipse plug-in code for the annotator
util.logging	Log4j integration in Eclipse

The prefix ch.ethz.inf.sct. is common for all plug-ins.

4.1.2. Identifier Naming Conventions for AST Elements

The class members that are represented in the AST have to be uniquely identifiable for some tasks. That is the lookup of local preferences, the determination of the coverage, and the identification of UtiVariables. In this context, an identifier is a string key that uniquely identifies a class member like a field, a parameter, or a return type. The identifiers have to be consistent for the Runtime and the Static Inference. Therefore, we introduce the following conventions how to build these identifier strings:

Fields

Grammar: (package-name ".")* class-name "." field-name

Example: com.acme.test.List.first

Parameter

Grammar: (package-name ".")* class-name "." method-name "(" (parameter-type (",")?)* = ")" ":PARAM" parameter-index
Example: com.acme.test.Node.insert(int,Object):PARAM0

Return Type

Grammar: (package-name ".")* class-name "." method-name "(" (parameter-type (",")?)* ")"
Example: com.acme.test.Node.remove(Object)

4.2. Command Line Tools

4.2.1. JVMTI Agent

Since there was no binary for the agent under Windows we tried to compile the agent source code with Microsoft Visual Studio. That required to replace the include files jni.h and jvmti.h with the ones provided by the Windows version of the Java SDK. The include files can be found in the include directory of your JDK installation. Unfortunately, the tracing agent crashed during the execution, even though the code was identical to the Linux version. The reason why it crashed was as follows:

The instrumenter class is loaded into a byte array. Under Linux the class was correctly loaded into the memory, but that was not the case under Windows. A memory access violation occurred because `file_stats.st_size` returned a larger size than the current length of the byte array in the memory. Opening the file in the mode `ios::binary` solved the issue.

```
ifstream inst_file;
inst_file.open(instrumenter_file_path, ios::binary);
jbyte* buf = NULL;
buf = new jbyte[file_stats.st_size];
inst_file.read((char*)buf, file_stats.st_size);
```

The instrumenter class, its location, and required library files were hard-coded in the tracing agent. We replaced these constants such that the tracing agent can be parametrized. We also added some exception handling to the agent code. Errors are shown on the standard output and the agent exits with one of the following exit codes:

0	normal termination of the agent.
-1	Unknown error.
-2	An argument is not set correctly.
-3	A class could not be found on the classpath
-4	The instrumenter class is not found

Usage

If the JVMTI agent is used in Eclipse, it can be started in the launch configuration (see the user guide in Chapter 3 for details). It is also possible to start the agent on the command line. In that case the usage of the JVMTI tracing agent is as follows:

```
java -cp <classpath> -agentpath:<ap>=<mc>,<of>,<ifp>,<icn>,<il> <mc>
```

where:

classpath: The classpath of the program to trace.

ap: The JVMTI agent is used by the Java Virtual Machine as a shared library. The path to the agent must contain the absolute path to the agent plus the agent name itself. For Linux systems the library `uts_type_inferer.so` is needed, for Windows systems the DLL `win32_trace_agent.dll`.

mc: The fully qualified name of the class containing the main method to execute. This includes the exact package name of the class. Do not omit to add this class to the classpath specified above! This information is needed twice for the following two reasons: first, the Java Virtual Machine has to know which program to start and second, the agent has to be informed about the main class to be able to start the tracing correctly.

of: The path to the file into which the gathered information will be written.

ifp: The *path* to the instrumenter class that is used for the bytecode instrumentation. The path must contain the absolute or relative path to the instrumenter class, including the file extension `.class`.
Example: `bin/ch/ethz/inf/sct/runtime_typinfer/instrument/Instrumenter.class`

icn: The fully qualified *name* of the instrumenter class. This includes the exact package name of the instrumenter class. The class has to be included in the classpath specified above.
Example: `ch/ethz/inf/sct/runtime_typinfer/instrument/Instrumenter`

il: Path(s) to JAR libraries which are needed for the bytecode instrumentation. Currently, we only need javassist.jar
Example: `lib/javassist.jar`

4.2.2. Java SE 6 Type Checking Verifier

The Java Virtual Machines provides means for a static analysis of the bytecode called *bytecode verification*. A bytecode verifier checks at loading time whether the program is well typed, only initialized variables are read, and some other properties. Until Java 5 the type information was inferred using a fixpoint iteration called abstract interpretation (see [23] for detailed description of the algorithms and formalizations). With Java SE 6, released in December 2006, Sun introduced a much simpler and faster verifier, called the *type-checking verifier*. The bytecode verification is split into two phases: the type inferencing and the type checking. Unlike the old verifier process the type inferencing is done during compile time and the type information to check the bytecode type consistency is stored in the class file. That enlarges the class file, but due to a simpler type checking algorithm at loading time the performance is improved by 50% [20].

The type-checking verifier required to change the class file format and added the `StackMapTable` code attribute [19]. The `StackMapTable` contains information like

- targets of conditional and unconditional jumps

- entry points of exception handlers
- instructions immediately following an instruction that unconditionally changes the control flow

The consequence of this change is that tools that manipulate the bytecode also have to adapt the new `StackMapTable` attribute correctly. The bytecode library javassist that we use to instrument class files for the tracing agent did not support this feature and, therefore, breaks the new type-checking verifier. We could not find official information about the support for JDK 6 in the latest javassist release version 3.4. The fact that there is a class implementing `StackMapTable` seems to indicate that the type checking verifier is supported. Nevertheless, Sun implemented a fall-back behaviour that allows to use the old verifier in the case the new verifier fails. This fall-back mechanism is controlled by the -XX command line flag `FailOverToOldVerifier` and is enabled by default. Make sure that this flag is enabled until a new version of javassist is released.

4.3. Annotator

The annotator is a tool that inserts the annotations stored in an annotion XML file into the Java source code. The first version of an annotation tool was implemented as a semester project by Marco Meyer [26]. We will show how we re-implemented the annotation tool to meet our requirements.

4.3.1. Design and Shortcomings of First Version

Design

The Annotation Tool mainly has three tasks to do. Firstly, we have to read in the annotations provided by the Type Inferer. Secondly, we have to read in the input Java source file. As a third task we have to combine the first two tasks so that the annotations get inserted into the source code.
The Java source files are parsed with a parser that is generated with JavaCC[1] and the Java Tree Builder JTB[2]. The Java Tree Builder JTB provides the Abstract Syntax Tree (AST) and an interface to the AST nodes using the visitor pattern. With a customized visitor the syntax tree nodes can easily be explored and modified. Visiting all AST nodes and output the syntax tree as an exact copy of the original input source is called "pretty printing". In addition to just dumping the unmodified source file the first version of the annotation tool extended the pretty printer by adding the Universe modifiers provided by the annotation XML at appropriate positions. The parsing of the annotations in XML file format is done by a direct mapping of the structured data into Java objects. As a binding technology the Apache XMLBeans[30] Framework is used.

Shortcomings

The annotation tool by [26] did not meet all our requirements. The tool did not accept pre-annotated source files and there was no support for the new Java 1.5 keywords.
The grammar file used for the Annotation Tool was the one delivered by JTB. There were no modifications done on this grammar file and the AST remained untouched as well. As

a consequence it was not possible to parse and process pre-annotated source files. This is crucial for our Static Inference integration since we need to be able to annotate pre-annotated files during the incremental Static Inference steps. Source files that were using the Java 1.5 standard were rejected because of the obsolete grammar file. The annotation tool had some problems with annotating nested structures like nested and anonymous classes.

4.3.2. New Implementation

Since the annotation tool was not Java 1.5 compliant, we modified the grammar file and re-implemented the whole annotator tool. The grammar should support the whole Java 1.5 syntax like generics or the sugared form of `for` loops. The grammar file is also modified to accept the Universe type modifiers **peer**, **rep**, and **readonly**. The modifiers are accepted in field declarations, local variable declarations, object instantiations, parameter declarations, return types, and cast expressions. Methods can be annotated with the keyword **pure** .
The Universe modifiers can be used in a JML-like style with a comment notation */*@ <jml–expression> @*/*. That means that we have to interpret the comments. And because we do not need the parser for a compilation task, but for a source code transformation, we would like to have the original comments in the transformed source code as well. Normally parsers simply ignore comments by filtering out the comment tokens in the tokenizer phase. The JavaCC parser uses a special concept for source code comments. Comments are handled as *special tokens* during the tokenizer phase. A special token does not participate in parsing. Instead, it is saved to be returned along with the next normal token. Thus, it is possible to access a special token through its normal neighbour token over the field `specialToken`. This feature of the JavaCC parser generator enables to add the comments when the AST is serialized with a pretty printer. Before parsing the source code the annotator executes some preprocessor steps. The input source is scanned and JML-style Universe type annotation patterns are replaced with the Universe keyword in the normal keyword style.

We also had to harmonize the annotation XML schema in order to have a consistent annotation format for both inference tools. Because some internals in the annotation XML format like indexing of parameters and assignments was not properly documented, we described that in Appendix C.

AnnotationFilterReader

Using the decorator pattern the input stream is scanned to detect JML-style Universe modifiers and replace these with the modifiers in keyword-style. This happens before the parsing. The preprocessor step simplifies the grammar since we do not have to treat the JML comments as special keywords and can handle it consistently like the normal keywords (without the comment notation). The `AnnotationFilterReader` is a normal `java.io.Reader`, so that the filtering is transparent to the user of the reader. The tokens in the stream are matched and replaced with the following regular expressions:

```
(/\*)\s*@\s*\\?(peer|rep|readonly)(\s*\\?(peer|rep|readonly))?[^\*/]*(\*/)
```

matches Universe modifiers for references.

```
(/\*)\s*@\s*(pure).*(\*/)
```

matches Universe modifiers for methods.

```
(//)\s*@\\?(peer|rep|readonly|pure)(\s*\\?(peer|rep|readonly))?.*
```

matches Universe modifiers in single comment lines.

JTBParser

The `JTBParser` class is the main class of the parser. It is generated by the parser generator JavaCC, implements a top-down parser, and builds the syntax tree provided by JTB. For more information about the generation of these files see Appendix A. The parser is started by passing an InputReader during construction and invoking its method `CompilationUnit`().

XMLAnnotationReader

The `XMLAnnotationReader` is used by the `AnnotationVisitor` and reads the annotation information from the XML annotation file. The `XMLAnnotationReader` is an intermediate file that does the traversing of the object graph and provides a simple interface to get the annotation information. The parsing and mapping of the XML is done by the Apache XML-Beans framework. The current implementation traverses the mapped object graph with each lookup of an annotation. If speed becomes an important factor, the performance of the `XMLAnnotationReader` could be improved by using a cache for the annotation lookup. Another improvement would be to query the XML by using XPath or XQuery.

AnnotationVisitor

The `AnnotationVisitor` is a vistor for the abstract syntax tree built up by the Java Tree Builder (JTB). It extends the `DepthFirstVisitor` which is, as the name implies, a depth-first traversal of the syntax tree. During visiting the AST nodes, the Universe annotation information is looked up from the XML file. New nodes for the annotations are created and added at the appropriate position in the tree for all field declarations, local variable declarations, object instantiations, parameter declarations, return types, and cast expressions. The top of the stack corresponds to the currently investigated class and method. Using a stack we can annotate nested and anonymous classes without any complications.
In addition to the visitor methods, the `AnnotationVisitor` implements some helper methods for creating the new AST nodes for the Universe modifiers. Since the AST structure can be quite long and complicated, the Eclipse debugging tools were a huge help during the development. The grammar file is not intuitive enough to imagine the AST structure with its long and nested branches. The interactive visualization given by the JDT debugger is a great tool for this job since we do not have a special AST view.

PrettyPrinter

The `PrettyPrinter` is another extension of the DepthFirstVisitor. The AST nodes are visited in order to dump the tree into a source file representation. The pretty printer provided in the JTB libraries only supported a Java 1.4 grammar without Universe modifiers. In order to support Java 1.5 and the Universe modifier keywords we had to write our own implementation of a pretty printer. The pretty printer adjusts the line and column information of each syntax

node. After the `PrettyPrinter` visited each node the syntax tree can be passed to the TreeDumper in order to dump the tree into a character stream.
The `PrettyPrinter` formats the source code according to the Java code conventions [18]. There might be slightly differences to the official conventions. We did not test that extensively.

ClassInformation, FieldInformation, InitializerInformation, MethodInformation

These are helper classes that keep track of class, field, initializer, and method information. This information is needed because the visitor method implementations are dependent on the grammar file and sometimes additional information from a different method is used in a certain context.

4.4. Logging in Eclipse

The Static and Runtime Inference stand-alone tools were command line applications. The interaction with a user happens with the standard input and output console. Logging events that are displayed on the standard output are useless in a GUI environment. They might appear in log files, but are not visible to the user in a simple manner. Fortunately, Apache log4j was used as a logging facility that helped a lot to integrate the existing code into Eclipse. This section gives a short overview about the logging tools in Eclipse and how we integrated the logging of the stand-alone tools into Eclipse. Our ideas about a log4j integration in our plug-ins were influenced by the article [25].

Eclipse Logging Tools

Eclipse provides its logging services with the interface `org.eclipse.core.runtime.ILog`. This interface is accessed through the method `getLog()` from the Plug-In activator class. Just create an instance of `org.eclipse.core.runtime.Status` with the right information and call the `log()` method on ILog. The Ready for IBM Rational software (RFRS) requirements indicate that exceptions and other service-related information should be appended to a log file and users are informed about log events. Therefore, the log object accepts more than one log listener instance. Eclipse adds two listeners, one that writes to the “Error Log” View and one that writes to the generic Eclipse log file. You can create your own log listeners as well. Just implement the interface `org.eclipse.core.runtime.ILogListener` and add the listener through the method `addLogListener()` to the log object. The listener is notified about each log event.

Log4j Integration

Log4j has the concept of a hierarchy of Logger objects that can send log events to any number of handlers (called Appenders in log4j), which delegate the message formatting to a formatter (called Layouts in log4j). One of log4j’s major strengths is that the tool is easy to extend. All we have to do is to implement a custom appender for log4j. For that appender, `org.apache.log4j.AppenderSkeleton` is extended and implemented in `ch.ethz.inf.sct.util.logging.PluginLogAppender`. The appender translates a `org.apache.log4j.spi.LoggingEvent` event into an Eclipse Status event. This status is passed to the plug-in’s `log()` method. The log4j error levels are translated into status instance codes the following way:

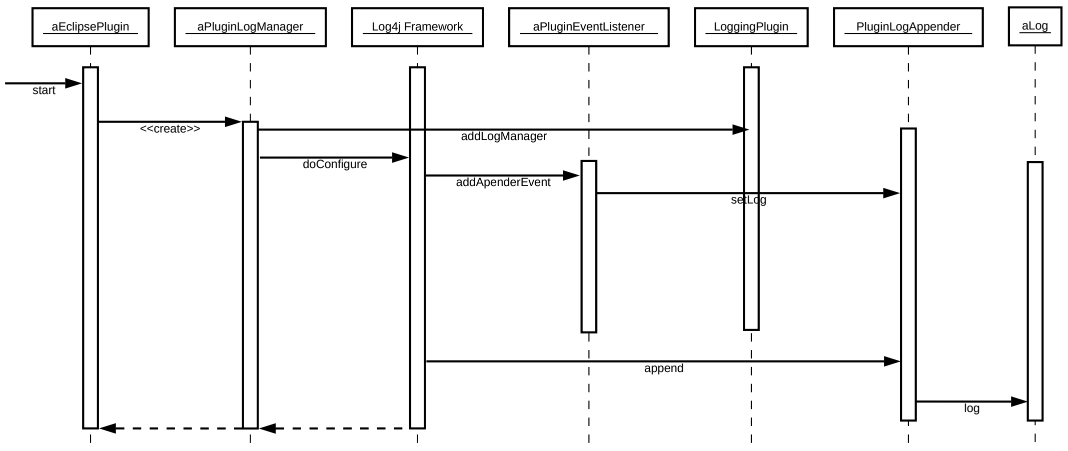


Figure 4.2.: The Log4j integration into the Eclipse logging facility.

- Level.FATAL → Status.ERROR
- Level.ERROR → Status.ERROR
- Level.WARN → Status.WARNING
- Level.DEBUG → Status.INFO
- default → Status.OK

Log4j is seen as a black box that may process several different tasks internally.

The whole log4j integration is implemented in a plug-in project called ch.ethz.inf.sct.util.-logging. To add logging support to a plug-in project, the plug-in writer just has to add ch.-ethz.inf.sct.util.logging to the dependency list and create a log4j configuration file. Instantiate a `PluginLogManager` and configure it with this file. This should be done only once, so you can do it when the plug-in starts. For the log statements, just use them as you would do with log4j. Listing 4.1 shows an example and Figure 4.2 explains the call sequences of the log4j integration.

Listing 4.1: PluginLogManager configuration inside a plug-in.

```
private static final String LOG_PROPERTIES_FILE = "logger.properties";

public void start(BundleContext context) throws Exception {
   super.start(context);
   configure();
}

private void configure() {
   try {
      URL url = getBundle().getEntry("/" + LOG_PROPERTIES_FILE);
      InputStream propertiesInputStream = url.openStream();
      if (propertiesInputStream != null) {
         Properties props = new Properties();
```

```
            props.load(propertiesInputStream);
            propertiesInputStream.close();
            this.logManager = new PluginLogManager(this, props);
            this.logManager.hookPlugin(
             TestPlugin.getDefault().getBundle().getSymbolicName(),
             TestPlugin.getDefault().getLog());
         }
      }
      catch (Exception e) {
         String message = "Error␣while␣initializing␣log␣properties." + e.getMessage();
         IStatus status = new Status(IStatus.ERROR,
         getDefault().getBundle().getSymbolicName(),
         IStatus.ERROR, message, e);
         getLog().log(status);
         throw new RuntimeException("Error␣while␣initializing␣log␣properties.",e);
      }
}
```

The `PluginLogManager` uses the properties from the property file (a normal log4j property file) and configures the log4j framework. The property file has to be configured in a way that the `PluginLogAppender` gets added as an appender to the log4j framework. For that purpose configure the appender in the property file as follows:

```
# A2 is set to be a PluginLogAppender
log4j.appender.A2=ch.ethz.inf.sct.util.logging.PluginLogAppender
log4j.appender.A2.layout=org.apache.log4j.PatternLayout
log4j.appender.A2.layout.ConversionPattern=%p %t %c - %m%n

# add appender A2 to ch.ethz.inf.sct.inferer.si level only
log4j.logger.ch.ethz.inf.sct.inferer.si=, A2
```

The log4j framework fires an `addAppenderEvent` which adds the current log instance to the `PluginLogAppender`. In our case the log instance (aLog in the sequence diagram) is the Eclipse log object. If the `PluginLogAppender` now gets notified from log4j about a log event with `append(LoggingEvent)`, it can transform and forward the event to the Eclipse log object. Finally, the event is shown in the Error Log view.

4.5. Runtime Inference Tool

Section 4.5.1 shows the design of the Runtime Inference Tool and some shortcomings that had to be fixed in order to integrate it into Eclipse. Because the source code of the tracing agent was not yet well documented, we briefly highlight some aspects of the agent that were important for us. How we solved and implemented the issues is shown in Subsection 4.5.2

4.5.1. Design and Shortcomings of the Runtime Inference Tool

Design

The tracing agent is loaded by the Java VM. The main function that is executed after attaching to the VM is `Agent_OnLoad(JavaVM *vm, char *options, void *reserved)`. This function checks whether all required arguments are passed and valid. After that the agent registers its callback functions to get notifications about the JVMTI events. Whenever a class is loaded, the agent is notified in the callback function `ClassFileLoadHookCallback`. The agent dynamically loads the Java class that is used to instrument bytecode (Instrumenter.class in the instrumenter package) and instruments the class that is currently being loaded by the VM. The instrumentation happens in-memory by operating on the bytecode array of the loaded class. After instrumenting, the modified bytecode array is returned back to the VM. So, we directly interact with the JVM class loader. `MethodEntryCallback` and `FieldModificationCallback` are the callback functions for field modification and method entry events. They directly print out the event with the corresponding arguments to the tracing XML file.

The runtime inference algorithm is implemented as a couple of visitor classes. The visitors operate directly on the EOG. The `BuildUpVisitor` builds the graph based on the tracing XML file. The `DominatorVisitor` computes the direct dominator for each graph node and sets the `owner` field of every `GObject` node. After that the `StoreDominatorLevelVisitor` computes the depth of each object in the dominator tree. `ResolveConflictsVisitor` is used to resolve conflicts that are introduced by the owner-as-dominator approximation. If enabled, the `AbstractInterpretationVisitor` performs the abstract interpretation of method bodies and the `HarmonizationVisitor` maps the dynamic EOG structure to a static structure of the program classes. Finally, the `OutputVisitor` writes the annotations into an annotation XML file.

Shortcomings

We experienced some issues with the tracing agent. Firstly, the agent did not compile under Windows. After having fixed these problems, the agent crashed the VM process. The resolution of this issue is described in Subsection 4.2.1. The agent implementation did also hard-code a lot of paths and constants which had to be parameterized.

The integration approach taken by Bazzi[5] ran the type inferer in an own Java VM process. This is not suitable for our purpose. Passing of inferer configurations in-memory or the interaction with the Inference Visualizer over the observer interface requires that the type inferer runs in the same VM process. Like in the Static Inference, process termination with `System.exit(−1)` is not acceptable because it terminates the whole Eclipse instance.

Further, the Runtime Inference uses Java reflection to dynamically load classes and read class information in the visitor implementations. In the preceding projects the type inferer ran in the same classpath as the test project did as well. Using Eclipse the project classes are not in the classpath automatically. How to resolve this issue is described in Subsection 4.8.1.

4.5.2. Eclipse Integration

The following enumeration picks out the most important classes of the Runtime Inference plug-in. The list is not complete, further documentation is available as javadoc source comments.

RuntimeInferenceLaunchDelegate [plugin package]

The `RuntimeInferenceLaunchDelegate` is the launch delegate for a Runtime Inference configuration. It creates the configuration out of the settings defined in the inference configuration dialog and passes the configuration to the RI inferer. If the visualizer plug-in is installed and the visualizer is enabled, the delegate opens the visualizer editor.

EclipseRIInterface [plugin.core package]

The EclipseRIInterface acts as an interface between the Runtime Inference classes in the RI resource plug-in and the controller (Eclipse in this implementation). The superclass `AInferenceController` divides the inference process in several subtasks that are executed in methods:

- **init():** Initialization of the inferer. Only called once.
- **beforeInfer():** Pre-inference tasks like setting up the visualizer and registering the trace observer(s).
- **infer():** The main inference tasks. Running the inferer backend.
- **afterInfer():** Post-inference tasks.

RIAgentLaunchListener [plugin.core package]

The `RIAgentLaunchListener` listens to RI tracing agent launches. As soon as the tracing agent VM process terminates, we are notified through a DebugEvent and update the project directory. The goal is to reflect the changes (adding or modifying a trace file) in the file structure of the project and navigator view (see Subsection 4.8.5 as well).

RIPlugin [plugin.core package]

`RIPlugin` is the main and activator class which controls the life cycle of the Runtime Inference plug-in. In addition to starting and stopping the plug-in and offering some helper methods it holds a reference to `PluginLogManager` to redirect log4j logging events to the Eclipse log view.

RIProjectPropertiesPage [plugin.ui package]

`RIProjectPropertiesPage` implements the UI components of the project property dialog for the Runtime Inference. The properties can be changed under the tab ”Runtime Inference” in the project properties. This class also handles the persistence of the properties.

Inference Configuration Tabs [plugin.ui package]

The class `RuntimeInferenceTabGroup` assembles the tabs used for RI inference configuration. The following tabs are settled in this package:

- **TraceTab:** The tab to add trace, purity file, and to define the annotation output file.
- **VisualizerTab:** The tab to enable or disable the visualizer.

4.6. Static Inference Tool

Section 4.6.1 roughly explains the implementation and shows all components of the Static Universe Type Inference designed by Matthias Niklaus[29]. We discuss some shortcomings of the design regarding an Eclipse integration. How these issues are solved and implemented is shown in Section 4.6.2

4.6.1. Design and Shortcomings of the Static Inference Tool

Design

The architecture that was chosen by the SUTI2 project is splitted up in two parts. The client part and the inferer part. The first part provides the Java program structure to the inferer. A parser parses the source code and builds the constraints used for the Universe type inference. The latter, the inferer part, takes all provided program information into account and infers an optimal solution for the Universe types. Between these two parts is a clearly defined interface, called **U**niverse **T**ype Inference **I**nterface (UTI), which allows the independent development of both parts.
The JML client provides the interface with the required information for the type inference. It parses the source file using the JML parser and typechecker of the MultiJava project. That is why the Java sources might be pre-annotated with Universe modifiers. From the JML parser we get the abstract syntax tree. The AST is traversed with the visitor `UniverseJmlVisitor`. This visitor translates the program structure provided by the AST to the necessary `UtiConstraintBuilder` interface method calls that build up the constraints. It also builds up the `JmlToUtiMapper` which is a mapping between UTI variables and the corresponding nodes from the JML AST.
After all information is provided, the inferring process may be started. This process is controlled by a controller. The controller creates a new process and runs the PBSTool.

Shortcomings

The architecture by [29] defined a separation between the model (UTI Interface), the view (`SimpleUI`), and the control (`UserInteraction`). A misuse of the MVC separation happens with the method `interact(UtiController, AnnotationWriter)` in the `UtiController` interface. That passes the control and the model to the view. The design of the `SimpleUI`, a mixture of view and control, was driven by the sequential command-line oriented process. A further issue of that design was the following: if the execution of the inference controller was interrupted due to an exceptional state, the process was stopped by calling `System.exit`(-1). This destroyed the instance of the Virtual Machine and stopped the program which is okay for a command-line tool. Using a graphical environment like Eclipse this behavior is not desirable

because stopping the virtual machine shuts down the whole Eclipse instance. Creating a new virtual machine process for each inference run is not possible because the constraints and result of the inference are used in an Eclipse view and require inter-process communication between different virtual machine instances. Also the performance of the inferer would suffer from creation of new processes each time.
The SUTI2 project seemed to work perfectly under Linux. Nevertheless, a test on a Windows machine did not work out. The parser crashed with a NullPointerException. It turned out to be an issue in the TypeLoader of the JML compiler that was originated by wrong path comparisons. The helper function `Helper.isEqualFilePath`() compared two paths by using a simple String comparison. Since Windows and Linux are using different path separators that led to the problem. The fixing of this bug was a work of two minutes even though the debugging took me several hours.
Another incompatibility was the tight coupling of the SUTI configuration with the XML configuration file. Using Eclipse it would be desirable to have convenient configuration settings in the project and workspace preferences without soiling one's hands with editing the XML configuration file. Therefore, the configuration interface had to be changed and decoupled from the XML instance.

4.6.2. Eclipse Integration

StaticInferenceController [plugin.core package]

The `StaticInferenceController` implements the controller, following a MVC pattern. It holds a reference the model (`AnnotationWriter`) and to the view (`UTAnnotationView`). The controller gets notified about changes in the model and the view through the handlers `modelChangedHandler` and `viewChangedHandler`. Also includes an implementation of User-Interaction which starts the user interaction after the inference is done by the inferer backend. The `EclipseUserInteraction` fires a `modelChanged` event, opens the `UTAnnotationView`, and displays the result of the executed inference step.

EclipseSIInterface [plugin.core package]

The `EclipseSIInterface` acts as an interface between the Static Inference classes and the controller (Eclipse in this implementation). The superclass `AInferenceController` divides the inference process into several subtasks that are executed in methods:

- **init():** Initialization of the inferer. Only called once.
- **beforeInfer():** Pre-inference tasks as setting the SolutionDescription or parsing the sources and building the AST.
- **infer():** The main inference tasks. Running the inferer backend.
- **afterInfer():** Post-inference tasks. Starts the interaction with the user interface after an inference step.

PreferencesList [plugin.core package]

The `PreferencesList` is the model of a list of preferences for the Static Inference. The preferences represent the weights that are used for the MAX-SAT formalization. The `PreferencesList` is needed for the persistence of the preferences values in a launch configuration for the Static Inference. The preferences are serialized and deserialized into a string list that can be handled by `ILaunchConfigurationWorkingCopy`. It is also possible to translate the preferences directly into an `UtiSolutionDescription`.

SIPlugin [plugin.core package]

`SIPlugin` is the main and activator class that controls the plug-in life cycle of the Static Inference plug-in. Besides starting and stopping the plug-in and offering some helper methods it holds a reference to `PluginLogManager` to redirect log4j logging events to the Eclipse log view.

SIProperties [plugin.core package]

Encapsulates the properties that are used for the Static Inference. Loads properties identified by a string identifier key from the persistent project property store. If properties are not available from the project property store it uses the default property settings defined in the file `staticinference.default.properties`.

StaticInferenceLaunchDelegate [plugin.launching package]

A launch delegate is the action delegate that is executed after clicking on the "Launch" button in the inference configuration dialog. We built our inference configuration on top of the launch configuration classes provided by Eclipse. As you might have noticed in the UI, the inference configuration dialog is identical to the launch configuration dialog. We extended Eclipse with a new type of launch configuration (extension point `org.eclipse.debug.core.launch-ConfigurationTypes`).
Similarly, the `StaticInferenceLaunchDelegate` is the extension of an `AbstractInference-LaunchConfigurationDelegate` and used for launching the Static Universe Type Inference. Also provides convenience methods for accessing and verifying launch configuration attributes such as the preferences and the `UtiSolutionDescription`.

SIProjectPropertiesPage [plugin.ui package]

`SIProjectPropertiesPage` implements the UI components of the property dialog for the Static Inference. The properties can be changed under the tab "Static Inference" in the project properties. This class also handles the persistence of the properties.

SIPreferencePage [plugin.ui package]

This class represents a preference page for the Static Inference preferences. It is used to edit the workspace settings for the Static Inference plug-in, such as the path to the SAT solver binary and the solver arguments.
The initial default values for the preference settings are provided by `PreferenceInitializer`.

UTAnnotationView [plugin.ui.views package]

The `UTAnnotationView` is a workbench view that shows the data obtained from the model `UTAnnotationWriter`. The view shows the inferred Universe Type annotations.

The `UTAnnotationView` displays a structured tree that shows the Universe types of the following members of a class

- Fields
- Field initializers
- Class initializers
- Casts
- Local variable definitions
- Methods
- Object creation
- Parameters
- Return types

Each kind of member is associated with an icon by `AnnotationViewLabelProvider`. The `UTAnnotationView` allows to set or prevent a certain annotation. After fixing or preventing a Universe type it is possible to re-infer the types and implicitly set the annotations of dependencies. The purity of methods can be changed as well. The `UTAnnotationView` provides actions to export the inferred annotations to and XML or to insert the annotations directly into the Java source code.

`UTAnnotationView` implements a `PageSelectionChanged` listener to be informed about selection events in other view parts. If a Java structure is selected in the source editor, the focus is set to the corresponding annotation structure in `UTAnnotationView`. A double click on an annotation member jumps to the corresponding source code in the Java editor.

AnnotationManager

The `AnnotationManager` is a singleton instance that holds a reference to the `AnnotationWriter` currently shown in the `UTAnnotationView`. Since a view part like `UTAnnotationView` is always a singleton instance in Eclipse, it is enough to keep a singleton instance of the model as well. Whenever a plug-in needs the last inferred Universe Type annotations it can get them by calling `AnnotationManager.getManager().getAnnotations()`. `AnnotationManager` also contains some helper methods that are related with the `AnnotationWriter` model.

4.7. Combined Inference

The Combined Inference tries to reuse as much as possible from the Static and the Runtime Inference. All the views and inferene configurations are shared. The Combined Inference plug-in adds all the new required functionality and data structures for the coverage and weight heuristics.

4.7.1. Coverage

The field and method coverage is calculated based on an annotation XML file. This is implemented in a class with the amazingly long name `AnnotationDocumentCoverageStructure Builder` which takes a parsed XML file (an `AnnotationsDocument`) and builds the coverage structure. The coverage structure is accessed over the class `Coverage`. It stores the coverage information and gets the values for the method, field, and parameter coverage. Because there is no guarantee about the order how classes appear in the trace file and therefore also no guarantee when they are added to `Coverage`, we have to build the class inheritance structure, the class – superclass relationships, at the end. The code has to ensure that all classes are added to the structure before a call to `getFieldCoverage`, `getNumFields`(), `getMethodCoverage` (), or `getNumMethods`() is made. `ClassCoverage` stores information about classes and its members, `FieldCoverage` about the fields. Method and parameter informations are kept in `MethodCoverage` and `ParameterCoverage` (see UML class diagram in Figure 4.3).

4.7.2. Weight Heuristics

WeightHeuristic

A WeightHeuristic is used to weight the suggested ownership annotations from the Runtime Inference. All heuristics must implement the interface `WeightHeuristic`. A `WeightHeuristic` is notified by a `WeightFactorStructureBuilder` about the events. Each field, method, or parameter triggers an event and lets the heuristic calculate the weight factor.

Weight heuristic can extend the abstract class `AWeightHeuristic` to share some common attributes and methods. At the moment there are two kind of heuristics. Those who are only based on the `WeightFactorStructure` and those who need additional parameters like a coverage heuristic that needs the coverage information. Future extensions may add more kinds of heuristic types. See the UML class diagram in Figure 4.4 about the inheritance structure.

How To Extend the Heuristics

The system can be extended with heuristics that are written by other contributors. We built a flexible extension mechanism for heuristics based on the Eclipse plug-in system. Other heuristics can be added as plug-ins to our Combined Inference plug-in. To do that one has to use the extension point ch.ethz.inf.sct.inference.ci.plugin.heuristic as follows:

1. Create a new Eclipse Plug-In project
2. Implement your heuristic class and implement the interface `WeightHeuristic`
3. Open the MANIFEST.MF in the PDE editor and add a new Extension for the extension point ch.ethz.inf.sct.inference.ci.plugin.heuristic
4. Add a new heuristic element and set the attributes as described in the schema below.
5. The heuristic is listed in the heuristic tab in the inference configuration for the Combined Inference (see 3.5).

Configuration Markup:

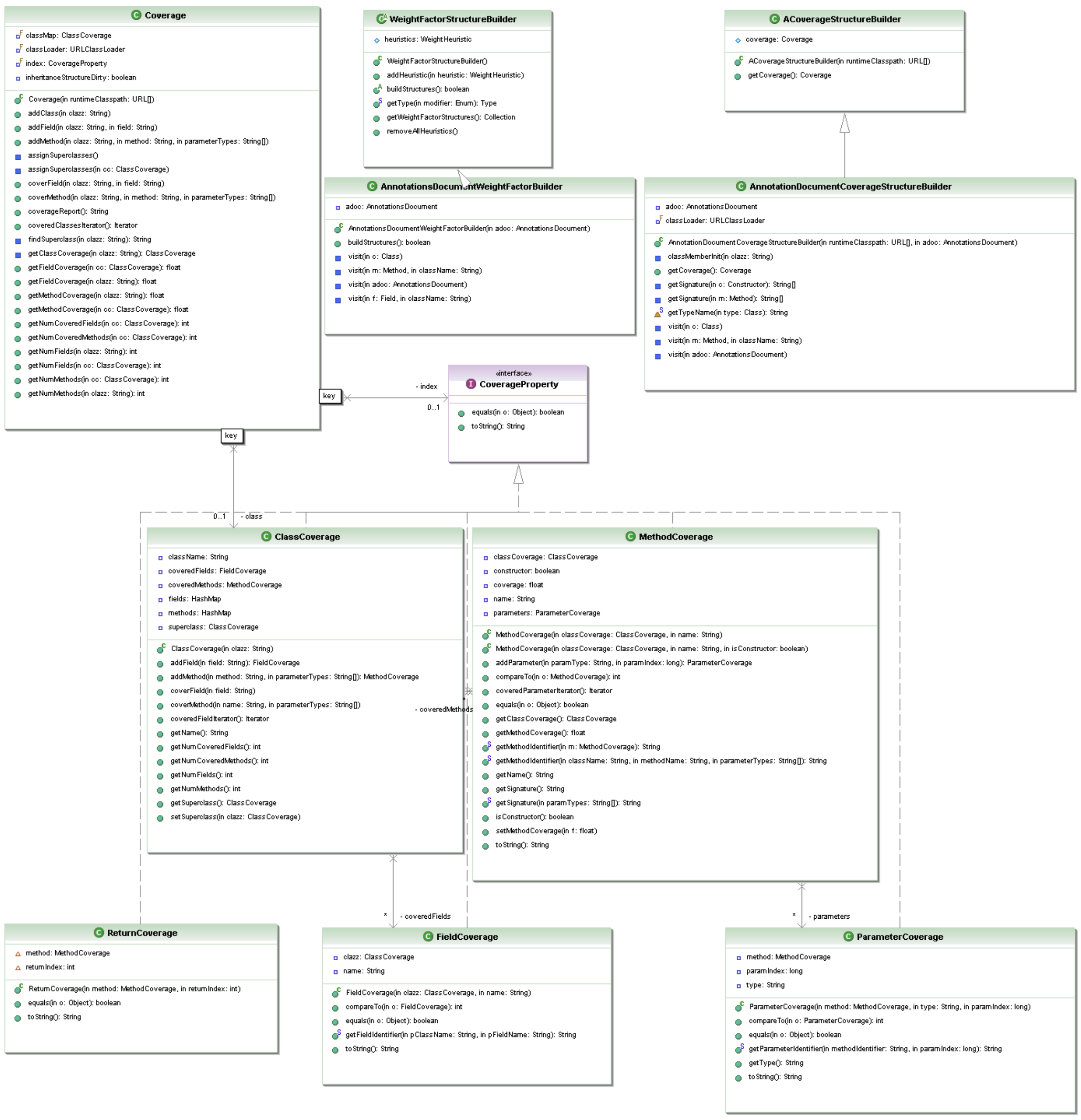

Coverage
classMap: ClassCoverage
classLoader: URLClassLoader
index: CoverageProperty
inheritanceStructureDirty: boolean
Coverage(in runtimeClasspath: URL[])
addClass(in clazz: String)
addField(in clazz: String, in field: String)
addMethod(in clazz: String, in method: String, in parameterTypes: String[])
assignSuperclasses()
assignSuperclasses(in cc: ClassCoverage)
coverField(in clazz: String, in field: String)
coverMethod(in clazz: String, in method: String, in parameterTypes: String[])
coverageReport(): String
coveredClassesIterator(): Iterator
findSuperclass(in clazz: String): String
getClassCoverage(in clazz: String): ClassCoverage
getFieldCoverage(in cc: ClassCoverage): float
getFieldCoverage(in clazz: String): float
getMethodCoverage(in clazz: String): float
getMethodCoverage(in cc: ClassCoverage): float
getNumCoveredFields(in cc: ClassCoverage): int
getNumCoveredMethods(in cc: ClassCoverage): int
getNumFields(in clazz: String): int
getNumFields(in cc: ClassCoverage): int
getNumMethods(in cc: ClassCoverage): int
getNumMethods(in clazz: String): int
WeightFactorStructureBuilder
heuristics: WeightHeuristic
WeightFactorStructureBuilder()
addHeuristic(in heuristic: WeightHeuristic)
buildStructures(): boolean
getType(in modifier: Enum): Type
getWeightFactorStructures(): Collection
removeAllHeuristics()
ACoverageStructureBuilder
coverage: Coverage
ACoverageStructureBuilder(in runtimeClasspath: URL[])
getCoverage(): Coverage
AnnotationsDocumentWeightFactorBuilder
adoc: AnnotationsDocument
AnnotationsDocumentWeightFactorBuilder(in adoc: AnnotationsDocument)
buildStructures(): boolean
visit(in c: Class)
visit(in m: Method, in className: String)
visit(in adoc: AnnotationsDocument)
visit(in f: Field, in className: String)
AnnotationDocumentCoverageStructureBuilder
adoc: AnnotationsDocument
classLoader: URLClassLoader
AnnotationDocumentCoverageStructureBuilder(in runtimeClasspath: URL[], in adoc: AnnotationsDocument)
classMemberInit(in clazz: String)
getCoverage(): Coverage
getSignature(in c: Constructor): String[]
getSignature(in m: Method): String[]
getTypeName(in type: Class): String
visit(in c: Class)
visit(in m: Method, in className: String)
visit(in adoc: AnnotationsDocument)
key
- index
0..1
«interface»
CoverageProperty
equals(in o: Object): boolean
toString(): String
key
0..1
- class
ClassCoverage
className: String
coveredFields: FieldCoverage
coveredMethods: MethodCoverage
fields: HashMap
methods: HashMap
superclass: ClassCoverage
ClassCoverage(in clazz: String)
addField(in field: String): FieldCoverage
addMethod(in method: String, in parameterTypes: String[]): MethodCoverage
coverField(in field: String)
coverMethod(in name: String, in parameterTypes: String[])
coveredFieldIterator(): Iterator
getName(): String
getNumCoveredFields(): int
getNumCoveredMethods(): int
getNumFields(): int
getNumMethods(): int
getSuperclass(): ClassCoverage
setSuperclass(in clazz: ClassCoverage)
- coveredMethods
*
MethodCoverage
classCoverage: ClassCoverage
constructor: boolean
coverage: float
name: String
parameters: ParameterCoverage
MethodCoverage(in classCoverage: ClassCoverage, in name: String)
MethodCoverage(in classCoverage: ClassCoverage, in name: String, in isConstructor: boolean)
addParameter(in paramType: String, in paramIndex: long): ParameterCoverage
compareTo(in o: MethodCoverage): int
coveredParameterIterator(): Iterator
equals(in o: Object): boolean
getClassCoverage(): ClassCoverage
getMethodCoverage(): float
getMethodIdentifier(in m: MethodCoverage): String
getMethodIdentifier(in className: String, in methodName: String, in parameterTypes: String[]): String
getName(): String
getSignature(): String
getSignature(in paramTypes: String[]): String
isConstructor(): boolean
setMethodCoverage(in f: float)
toString(): String
*
- coveredFields
*
- parameters
ReturnCoverage
method: MethodCoverage
returnIndex: int
ReturnCoverage(in method: MethodCoverage, in returnIndex: int)
equals(in o: Object): boolean
toString(): String
FieldCoverage
clazz: ClassCoverage
name: String
FieldCoverage(in clazz: ClassCoverage, in name: String)
compareTo(in o: FieldCoverage): int
equals(in o: Object): boolean
getFieldIdentifier(in pClassName: String, in pFieldName: String): String
toString(): String
ParameterCoverage
method: MethodCoverage
paramIndex: long
type: String
ParameterCoverage(in method: MethodCoverage, in type: String, in paramIndex: long)
compareTo(in o: ParameterCoverage): int
equals(in o: Object): boolean
getParameterIdentifier(in methodIdentifier: String, in paramIndex: long): String
getType(): String
toString(): String

Figure 4.3.: UML class diagram of the coverage model.

```
<!ELEMENT extension (heuristic)>
<!ATTLIST extension
  point CDATA #REQUIRED
  id    CDATA #IMPLIED
  name  CDATA #IMPLIED>

<!ELEMENT heuristic EMPTY>
<!ATTLIST heuristic
  id          CDATA #REQUIRED
  name        CDATA #REQUIRED
  description CDATA #REQUIRED
  type        CDATA #IMPLIED
  class       CDATA #REQUIRED>
```

id	The unique identifier for this weight heuristic.
name	The name of the heuristic
description	Textual description about the heuristic functionality.
type	[optional] The type/kind of weight heuristic (to distinguish different types of heuristic that need additional parameters like classes that extend the Coverage-Heuristic).
class	The class that implements a weight heuristic.

WeightFactorStructureBuilder

A `WeightFactorStructureBuilder` builds a `WeightFactorStructure` using different `Weight Heuristic`. The `WeightHeuristics` can be added to the `WeightFactorStructureBuilder` by calling `addHeuristic` and are notified by the structure builder. The collection of built WeightFactorStructures can be retrieved by calling `getWeightFactorStructures`().
The `AnnotationsDocumentWeightFactorBuilder` is an implementation of a `WeightFactor StructureBuilder` that builds the factor structure based on the information in an `Annotations Document`, a Java object representation of an annotation XML. This structure builder can be seen as the main interface between the Runtime and the Static Inference. It maps the information in the annotation XML into weights that can be used for the Static Inference.

4.8. Eclipse Integration Issues and Comments

This section discusses some general Eclipse integration issues that we have experienced in our work.

4.8.1. Class Loading

The `BuildUpVisitor` uses dynamic class loading for the inspected classes during the Runtime Inference. The classes are loaded dynamically to determine superclass relationships, settings

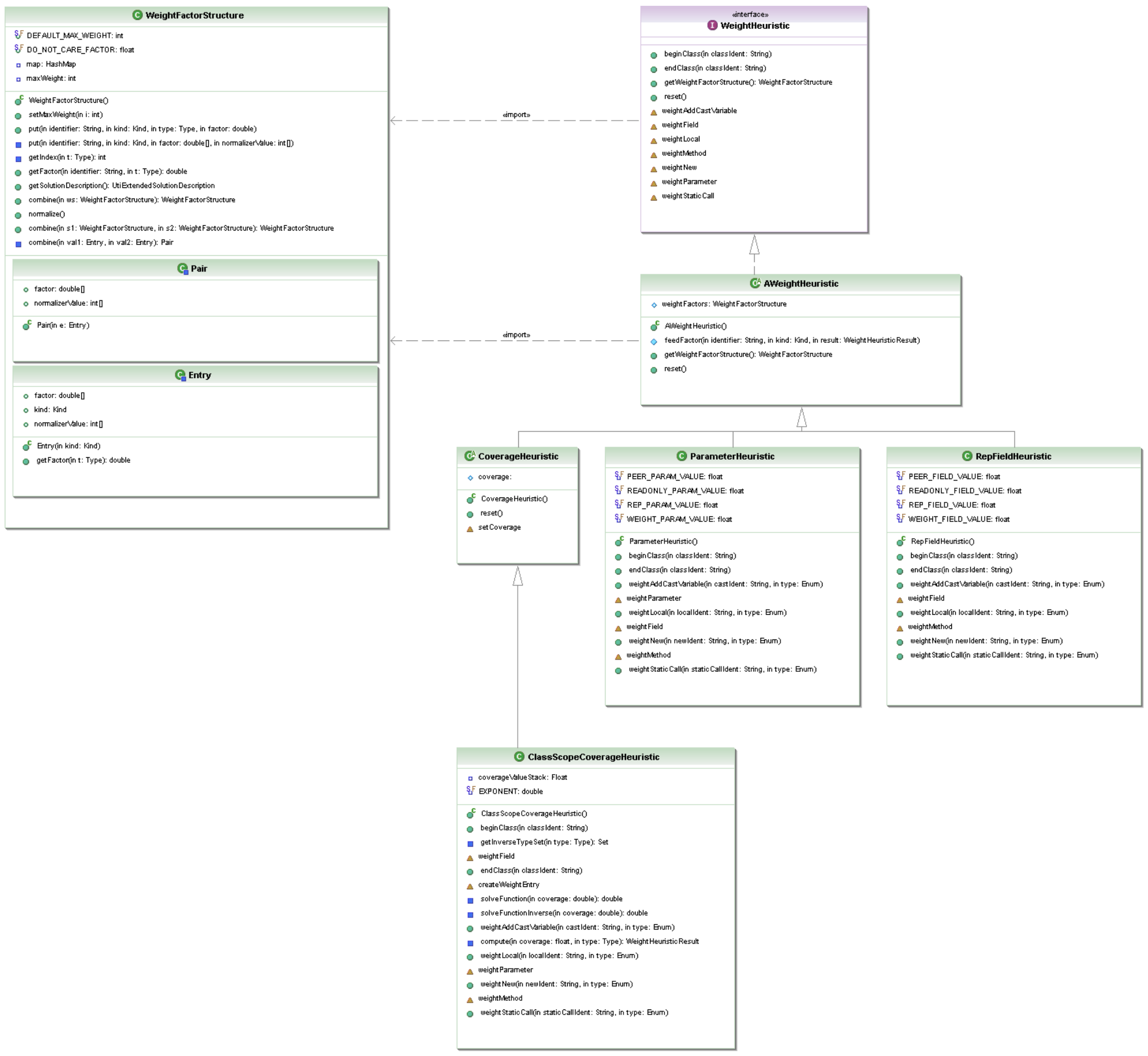

WeightFactorStructure
DEFAULT_MAX_WEIGHT: int
DO_NOT_CARE_FACTOR: float
map: HashMap
maxWeight: int
WeightFactorStructure()
setMaxWeight(in i: int)
put(in identifier: String, in kind: Kind, in type: Type, in factor: double)
put(in identifier: String, in kind: Kind, in factor: double[], in normalizerValue: int[])
getIndex(in t: Type): int
getFactor(in identifier: String, in t: Type): double
getSolutionDescription(): UtiExtendedSolutionDescription
combine(in ws: WeightFactorStructure): WeightFactorStructure
normalize()
combine(in s1: WeightFactorStructure, in s2: WeightFactorStructure): WeightFactorStructure
combine(in val1: Entry, in val2: Entry): Pair
Pair
factor: double[]
normalizerValue: int[]
Pair(in e: Entry)
Entry
factor: double[]
kind: Kind
normalizerValue: int[]
Entry(in kind: Kind)
getFactor(in t: Type): double
«import»
«import»
«interface»
WeightHeuristic
beginClass(in classIdent: String)
endClass(in classIdent: String)
getWeightFactorStructure(): WeightFactorStructure
reset()
weightAddCastVariable
weightField
weightLocal
weightMethod
weightNew
weightParameter
weightStaticCall
AWeightHeuristic
weightFactors: WeightFactorStructure
AWeightHeuristic()
feedFactor(in identifier: String, in kind: Kind, in result: WeightHeuristicResult)
getWeightFactorStructure(): WeightFactorStructure
reset()
CoverageHeuristic
coverage:
CoverageHeuristic()
reset()
setCoverage
ParameterHeuristic
PEER_PARAM_VALUE: float
READONLY_PARAM_VALUE: float
REP_PARAM_VALUE: float
WEIGHT_PARAM_VALUE: float
ParameterHeuristic()
beginClass(in classIdent: String)
endClass(in classIdent: String)
weightAddCastVariable(in castIdent: String, in type: Enum)
weightParameter
weightLocal(in localIdent: String, in type: Enum)
weightField
weightNew(in newIdent: String, in type: Enum)
weightMethod
weightStaticCall(in staticCallIdent: String, in type: Enum)
RepFieldHeuristic
PEER_FIELD_VALUE: float
READONLY_FIELD_VALUE: float
REP_FIELD_VALUE: float
WEIGHT_FIELD_VALUE: float
RepFieldHeuristic()
beginClass(in classIdent: String)
endClass(in classIdent: String)
weightAddCastVariable(in castIdent: String, in type: Enum)
weightField
weightLocal(in localIdent: String, in type: Enum)
weightMethod
weightNew(in newIdent: String, in type: Enum)
weightStaticCall(in staticCallIdent: String, in type: Enum)
ClassScopeCoverageHeuristic
coverageValueStack: Float
EXPONENT: double
ClassScopeCoverageHeuristic()
beginClass(in classIdent: String)
getInverseTypeSet(in type: Type): Set
weightField
endClass(in classIdent: String)
createWeightEntry
solveFunction(in coverage: double): double
solveFunctionInverse(in coverage: double): double
weightAddCastVariable(in castIdent: String, in type: Enum)
compute(in coverage: float, in type: Type): WeightHeuristicResult
weightLocal(in localIdent: String, in type: Enum)
weightParameter
weightNew(in newIdent: String, in type: Enum)
weightMethod
weightStaticCall(in staticCallIdent: String, in type: Enum)

Figure 4.4.: UML class diagram of the weight heuristic model.

of types, and other minor tasks with the Java reflection API. The implementation by [4] used the normal class loading mechanism with `Class.forName(String)`. This was fine because the examples were in the same runtime classpath like the Runtime Inference package. If you use `Class.forName(String)` in an Eclipse plug-in, then you will only be able to instantiate objects already known to your plug-in due to the usage of the plug-in's class loader and thus will only instantiate objects in your plug-in's classpath. What we need is a class loader that instantiates classes from the project folder. We fixed that by implementing our own extension of an `URLClassLoader`. The URLs added to the customized class loader have to be of the form `file:/path/to/a/directory/` and end with a slash. If the URL points to a file (without a slash at the end) it is assumed to point to a JAR file.

4.8.2. Insert Text into an Editor

The annotator uses its own parser and does not respect the AST built by the JDT editor. Using the Eclipse refactoring functionality would need a mapping from our own AST into the Eclipse representation. Regardless, to use the undo functionality of Eclipse we want to replace the whole text in the editor with our transformed source code. Text editors have no public API to insert text. Furthermore, they do not expose their internal widget used to edit the document. Therefore, inserting text in the currently active editor is not trivial. We solved this issue the following way:

Text editors obtain a document model by using a document provider. What we have to do is to locate the active editor, get its document provider, request the underlying document, and insert the source into it:

```
IWorkbenchPage page = ...;
IEditorPart part = page.getActiveEditor();
if (!(part instanceof AbstractTextEditor)
    return;
ITextEditor editor = (ITextEditor)part;
IDocumentProvider dp = editor.getDocumentProvider();
IDocument doc = dp.getDocument(editor.getEditorInput());
try {
    doc.replace(0, doc.getLength(), modifiedSource);
} catch (BadLocationException e) {
    ...
}
```

4.8.3. Accessing UI Thread

SWT is organized such that all drawing operations occur only in a special thread, called the UI thread. Any method that accesses or changes a SWT widget must be called in the UI thread. If not done this way, an `SWTException` with the value ERROR_THREAD_INVALID_ACCESS will be thrown.

We can access an SWT method in a non-UI thread in a safe manner by requesting that the default SWT display runs our runnable. This can be done as follows:

```
Display.getDefault().asyncExec(new Runnable() {
    public void run() {
            // access the SWT methods here
        }
    });
```

Using `Display.asyncExec(Runnable)` performs an asynchronous execution. This adds a request to the queue of the UI thread and returns immediately. If you need synchronous behavior instead, you can use `Display.syncExec(Runnable)`, which will suspend the current thread until the drawing took place in the UI thread.

Another possibility is subclassing `org.eclipse.ui.progress.UIJob`. The thread's run method `runInUIThread()` will, as the name implies, always run in the UI thread. It is also possible to install a listener on the job to find out when it completes and to set priorities for the job. Make sure that you never schedule jobs that take a long time to run. The UI thread will make the UI unresponsive if long running tasks are performed.

4.8.4. Obtaining a Workbench Reference

A frequently encountered issue is getting a reference to an `IWorkbechWindow` or `IWorkbenchPage`. A lot of APIs, like opening an editor or a view, are accessible over these objects. In the best case the following code snippet leads to the goal:

```
    IWorkbench workbench = PlatformUI.getWorkbench();
    IWorkbenchWindow window = workbench.getActiveWorkbenchWindow();
    IWorkbenchPage page = window.getActiveWorkbenchPage();
```

However, as documented in the Javadoc, these methods will return `null` if we do not have an active window (e.g., if we are in a non-UI worker thread). This means that another dialog or shell has the focus and we are not able to access these APIs to get the active window or page. As well, it returns `null` if we are in a non-UI worker thread. Have a look at 4.8.3 to see how you can schedule these operations into the UI thread.

There are several ways to obtain a workbench reference:

- If you are in a view or editor, you can do the following:

  ```
    IWorkbenchPage page = getSite().getPage();
  ```

- If you are in an action that implements `IWorkbechWindowActionDelegate`, you can get a reference to the window over the init method:

  ```
  class MyAction implements IWorkbenchWindowActionDelegate {
      private IWorkbenchWindow window;
          ...
      public void init(IWorkbenchWindow window) {
          this.window = window;
      }
  }
  ```

- If you are in a wizard, you can get access to the window over the init method of `IWorkbenchWinzard` similarly as above.

4.8.5. Process Termination Notification

The tracing agent runs in an own JVM process and not in the same VM as Eclipse. We had the problem that we did not know when the agent terminated and could not update the project directory or print out status messages about the agent's termination. Finally, we solved this issue by implementing the `RIAgentLaunchListener`.

Eclipse offers the interface `LaunchListener` that gets notifications in `launchChanged(ILaunch)` when a process or debug target is added to a launch instance. During the start of a launch configuration we have to register the launch listener in the `LaunchManager` of the Java Debug plug-in. Unfortunately, a `LaunchListener` does not get notifications when the VM process is terminated. For this purpose, we use the interface `IDebugEventSetListener` and implement the method `handleDebugEvents(DebugEvent[])`. That notifies us about DebugEvents. We look for the debug termination event `DebugEvent.TERMINATE`. If we get such an event, we compare whether the process that is the source of the debug event is equal to the process of our tracing agent launch instance. If yes, we know that the tracing agent has been terminated. The `LaunchListener` can now be deregistered from the `LaunchManager`.

4.9. Universe Visualizer

This section discusses the architecture and implementation of the Universe Visualizer. A prototype implementation of the Visualizer was done in [26]. Unfortunately, we were not able to successfully run this project and the architecture did not meet our requirements. Also, Meyer focused on the Runtime Inference visualization therefore his model was bound to an EOG. Since we also use the Universe visualization for the Static Inference, we introduced a more generic model. Our architecture differs significantly from Meyer's work.

Subsection 4.9.1 gives an overview and a very short introduction to GEF. A slightly deeper insight into the architecture is given in Subsection 4.9.2. The implementation is discussed in 4.9.3 and rounded off with the package overview in 4.9.5.

4.9.1. GEF Introduction

The Graphical Editing Framework(GEF) allows to easily develop rich graphical editors in Eclipse. All graphical visualization is done via the Draw2D framework, which is a standard 2D drawing framework based on SWT.

GEF: Separation of Concerns

GEF follows a strict MVC (model-view-controller) approach. The controller is the main part responsible for updating the view and performing model operations that are requested by UI events. GEF realizes the MVC separation as follows:

Model The model defines the data that gets visualized in an abstract way. Almost any arbitrary model can be implemented by a user. In our case, the model is the Universe structure (Universes and Classes/Objects). In our implementation the model notifies the controller about modifications.

View The view is anything visible to the user. It consist of figures and connections. Draw2D provides basic figures like rectangle, circles, but also more complex widgets like labels, buttons, or checkboxes.

Controller GEF usually has a controller for each visualized model. As mentioned in the introduction, the controller, called EditPart in GEF, is the link between the model and the view.

GEF Terms

Command Commands are used to edit the model. They describe how the model is modified in a way that can be undone and redone by the user.

Request Requests are the units of interaction in GEF. They are used to create, move, resize, delete, and group graphical objects in the view. A request contains information that might be necessary for executing the request later. An EditPart forwards a Request to an EditPolicy.

EditPolicy An EditPolicy understands a Request and creates a Command for it. Executing the Command will perform the model modifications needed to fulfill the Request.

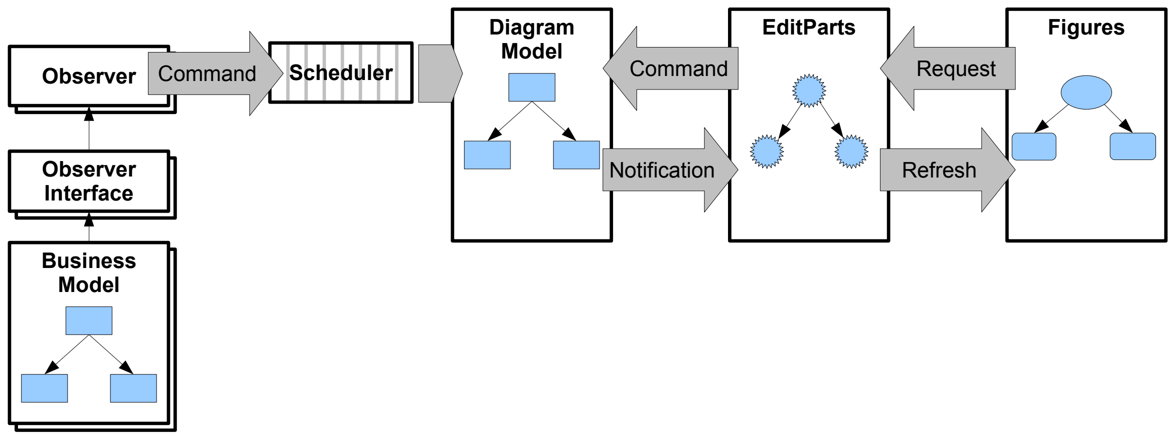


Figure 4.5.: The Inference Visualizer architecture overview.

4.9.2. Architecture

Figure 4.5 shows the overall architecture of our Inference Visualizer implementation. Notice the difference between the two models, called “Business Model” and “Diagram Model” in the figure. While the business model is the model that is built by the Universe type inferer, the EOG for the Runtime Inference and the AnnotationComponent structure for the Static Inference, the diagram model is used for modelling the graphical parts. The graphical model can abstractly be seen as a normal graph consisting of nodes and edges. It would be nice to have just one consistent model. Regarding the fact that the already existing models for the Runtime and Static Inference are different, we decided to do this step over an intermediate model.

Runtime Inference Visualization

The observer interface for the Runtime Inference is defined in `TraceObserver`. The observer itself can be found in `EOGObserver` in the visualizer plug-in. We extended the `TraceObserver` interface that was originally defined by [24]. We added two methods for the notifications when a trace observing starts and ends:

```
/** Notification about the start of tracing for the specified ProgramTraces */
public void tracingStarted(ProgramTraces traces, TraceVisitor changer);

/** Notification about the completion of traces. */
public void tracingCompleted(ProgramTraces traces, TraceVisitor changer);
```

All observer notification calls are executed with synchronous calls in the thread in which the inference algorithm runs. It enables to suspend the algorithm execution until the observer has finished the processing of the actual observer notification. This behavior allows the user to step through each action of the algorithm. The suspending of the algorithm thread takes place in the `Scheduler`. The scheduler is used for another purpose as well. GEF is built on top of SWT and interacts with the UI. Therefore, all GEF paint actions have to take place

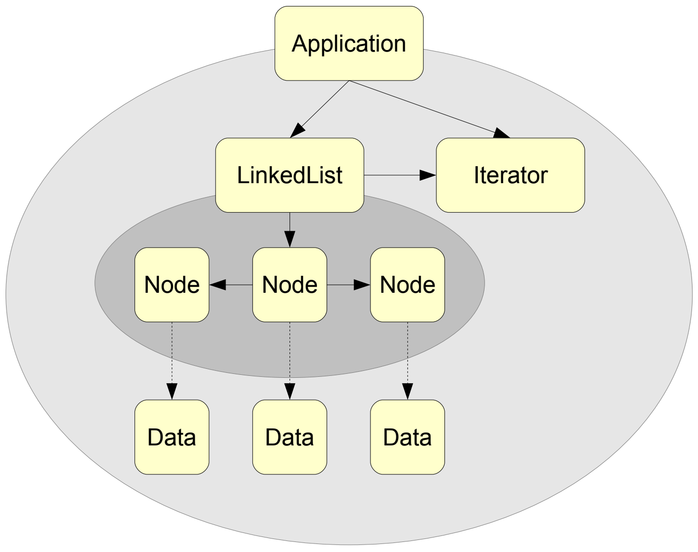


Figure 4.6.: A hypothetical visualization of the SI AnnotationComponent.

in the SWT UI thread. We use the scheduler as the synchronizer between the algorithm and the UI thread.

Static Inference Visualization

The visualization of the Runtime Inference is actually a visualization of the heap structure. An object node is generated for each object instance on the heap. The Static Inference visualization is not supposed to visualize instances, but the static class structure. Nevertheless, we would like to see an enhanced structure that is more suited for Universe visualizations.

Let us consider the doubly-linked list example in Listing 2.2 where the `previous` and `next` node are annotated as **`peer`**. A visualization is supposed to look like in Figure 4.6. While "normal" classes, such as `Application`, `LinkedList`, or `Iterator`, are just displayed once, we have to be careful with recursive structures like the `Node`.

We suggest to use the following algorithm:

```
Start with the main class cMain
visit(cMain)

function GraphNode visit(Class c) {
    if c in classtable of current universe and c is processed
        return corresponding node n for c from classtable
    if c contains rep field or rep in method signature
        create universe node n and store it in classtable of current universe
    else
        create class node n and store it in classtable of current universe

    if c is not yet processed {
        mark c as processed
        for each field in c {
            fnode = visit(f)
            create reference from n to fnode
        }
        for each parameter p in c {
            pnode = visit(p)
            create reference from n to pnode
        }
        for each return type r in c {
            rnode = visit(r)
            create reference from n to rnode
        }
    }
    return n
}
```

We visit all fields of a class exactly once. This guarantees the termination of the recursion. Notice that, applying the algorithm, the `Data` object is not displayed three time like in the figure, but only once.

Due to lack of time we were not able to implement the whole algorithm. Our current algorithm slightly differs from the one above. Only fields are considered. Support for parameters, return types, or local variables is left to future work.

4.9.3. Implementation

Model

All model objects must sublass `ModelElement` (see `model` package). `ModelElement` implements the basic functionality for the persistence and event notification mechanism described below.

Persistence is implemented using the standard Java object serialization into a binary format. All model elements implement `Serializable`. Fields that must not be serialized have to be annotated with the modifier `transient`.

We implemented an event notification mechanism in our model, although GEF itself does neither provide nor require such a functionality. The motivation is to easily be able to keep track of model changes in the EditParts. Whenever our model is changed, we fire an event that allows the EditPart to handle the model change and update the view accordingly. Our approach is to use the Java beans event support provided by the classes `java.beans .PropertyChangeSupport` and `java.beans.PropertyChangeListener`. Events are fired for the following model modification effects:

- **Children:** The hierarchy structure of our Universe graph changed. Children were added or removed from a Universe.
- **Owner:** The owner of an object changed.
- **Input:** Incoming references added or removed.

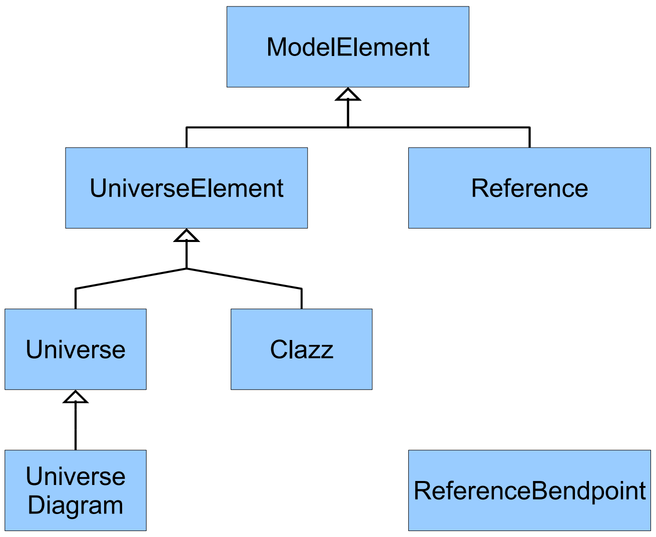


Figure 4.7.: The model of our Universe visualization

- **Output:** Outgoing reference added or removed.
- **Position:** The position (x-y-position) of a Universe element changed.
- **Layout:** The layout mode changed.

The owner changed event does not mean that we enhanced the Universe type system with ownership transfers. We need this event because the owner of a class or Universe might not be known during the build up of the EOG in the Runtime Inference and gets changed during the execution of the algorithm.

`UniverseElement` models all elements that are needed in the Universe structure visualization. It is subclassed by `Universe` and `Clazz`. While `Clazz` models a normal class, or in the case of the Runtime Inference visualization also object instances, a `Universe` is a container that can contain nested `Universes` and `Clazzes`. `UniverseDiagram` is a special kind of Universe, namely the root Universe. It is the root of our Universe structure and, therefore, also the root of a Universe diagram visualization.

`Reference` models references between objects and classes. Precisely, a `Reference` is a directed edge between two `UniverseElement`. A `ReferenceBendpoint` is a point through which a `Reference` has to be routed. Refer to Section 4.9.4 for more details about layouts and routing.

Controller

We use a factory to create `EditPart` objects. Typically this happens in the `PartFactory` when a creation command is executed. The factory has to be registered to GEF for that purpose.

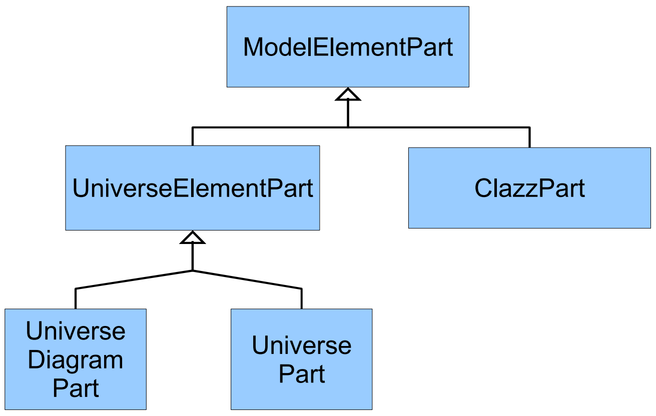


Figure 4.8.: The EditPart controller structure.

There are two types of `EditParts` that we implement by subclassing the corresponding abstract base classes provided by the framework: `GraphicalEditParts` and `ConnectionEditParts`. The former is used to represent model object with figures as graphical representation, the latter represents connections between `GraphicalEditParts`.

The `GraphicalEditParts` need to create the figure, update it on model changes, and dispose it if the `EditPart` is deactivated. The figure is created by overriding the method `createFigure`(). This method is called only once and the figure will be cached. Therefore, updating the figure has to take place in a dedicated method, namely `refreshVisuals`(). Figures have to be updated in this method implementation according to the encountered model changes.

The inheritance structure of the `EditParts` is shown in Figure 4.8. `ModelElementPart` implements the event notification handling. Normally, an event notification requests to redraw the figures by invoking `refreshVisuals`().

View

Everything that is visible in our Inference Visualizer plug-in is drawn in a figure. The figures implement the following functionality:

- Managing the figure hierarchy, adding and removing of child figures
- Accessor methods for layout managers (providing figure's preferred size and location)
- Setting the focus of a figure
- Painting
- Validating

`ClassFigure` implements the yellow class or instance boxes (see Figure 3.12 on page 56). It contains `CompartmentFigures` to add field and method labels to our class figures. That enables to visualize classes like in UML class diagrams.

A `SubgraphFigure` is used to paint container figures that can contain other nested figures. The `SubgraphFigure` consist of three parts, a header, a content, and a footer part. `UniverseFigure` subclasses `SubgraphFigure` and adds the owner class in the header part.

4.9.4. Layout

LayoutManagers are used to manage the position and size of figures. Respecting each figures preferred size, a layout algorithm calculates final size and locations of figures. Additional guidance for placements can be passed to the LayoutManager using constraints. It depends on the type of layout manager whether these constraints are respected or not. The Draw2D frameworks provides the following layout managers:

- **ToolbarLayout:** Arranges figures in a single row or column.
- **FlowLayout:** Lays out children figures in rows or columns, wrapping when the current row/column is filled.
- **GridLayout:** Lays out figures in a grid. Similar to the Swing GridLayout.
- **BorderLayout:** Similar to the Swing BorderLayout.
- **XYLayout:** Figures are placed at specified x-y-coordinates.

None of the built-in layouts meets our requirements exactly. Universe graphs can have quite a lot of references leading to confusing graph representation without a good layout mechanism. On one hand, we would like to have a layout that automatically places the nodes during the graph build-up. On the other hand, we would like to have manual editing functionality for the graph.

Our conclusion was to implement both approaches using different layout modes. We use a `DelegatingLayoutManager` that implements the whole `LayoutManager` interface and pretends to the EditParts to be a layout manager. In reality it just delegates the layouting tasks to `GraphLayoutManager` or `GraphXYLayoutManager` according to the chosen layout mode. `GraphLayoutManager` is the automatic layout that tries to place graph nodes and edges in an optimal way. `GraphXYLayoutManager` is the manual layout that lets the user move class and Universe elements. Changing the layout mode fires the Layout property in the model. That makes the controller change the layout manager in the `DelegatingLayoutManager` and request all figures to be repainted.

The automatic `GraphLayoutManager` uses the `DirectGraphLayout` implementation done by the GEF contributor Randy Hudson. It uses a network simplex algorithm for assigning ranks to the nodes. The nodes are ordered according to the ranks to minimize crossings of edges. The rank ordering is done using the Sugiyama algorithm[32]. The y-coordinates of each node is assigned based on the ranks. Final x-coordinates are assigned to a node using an auxiliary graph as described in [15].

The manual `GraphXYLayoutManager` is a simple layout that sets the location and size according to user interaction in the view. Moving or resizing an element in the view creates a request that is delegated to `UniverseXYLayoutPolicy`. The policy then creates a `ModelElementMoveCommand` command and sets the new layout constraints.

4.9.5. Package Overview

Action Package [ch.ethz.inf.sct.inference.visualizer.actions] The action package contains all classes that are related to actions and can be triggered through the user interface. `VisualizerActionBarContributor` contributes the actions as icons to the toolbar. These are the actions to control the player (`PlayAction`, `OneStepAction`, `PauseAction`) and `ChangeLayoutAction` to toggle between the automatic and the manual layout. `VisualizerContextMenuProvider` is the contributor for actions that appear in the context menu of the editor.

Editor Package [ch.ethz.inf.sct.inference.visualizer.editor] The `InferenceVisualizerEditor` implements the editor which is used for the Universe visualization. The editor initializes different objects like the `Scheduler`, the `GraphicalViewer`, and `OverviewOutlinePage`. It also creates the `VisualizerContextMenuProvider` and the ZoomManager for the toolbar and context menu contributions.

Eclipse editors are following an open-save-close model. That means, each editor instance corresonds to a file and is responsible to open, save, and close the associated file. Normally, this model also implies the following: Before opening a resource, the resources is already created and persistent on the hard disk. Therefore, an editor always needs an `IEditorInput`. Since we are starting our visualization by launching an inference configuration, we do not have this preexisting resource file to be passed to the editor. We solved this issue by creating an artificial input object, called `NonExistingFileEditorInput`, that is not related to a file. Once we save the file, we detect the missing file association and create the file on-the-fly. Saving a file serializes the GEF model. Therefore, the GEF model can be reconstructed at later time. This is only true for the graphical model. The EOG model is not serialized, consequently, the inferring process is interrupted.

`OverviewOutlinePage` is shown in the Outline View when a visualizer editor is activated. It shows the overview of the Universe graph in a thumbnail image.

Figures Package [ch.ethz.inf.sct.inference.visualizer.figures] The figure package contains all figures. See the description in the View subsection on page 87.

Layout Package [ch.ethz.inf.sct.inference.visualizer.layout] The layout package contains all layout related classes, such as the `DelegatingLayoutManager`, `GraphLayoutManager`, and the `GraphXYLayoutManager`. See the description in the Subsection 4.9.4.

Model Package [ch.ethz.inf.sct.inference.visualizer.model] The model package contains all classes that are related to the GEF diagram model. See the description in the Model subsection on page 85.

The subpackage *commands* contains all Commands used to modify the GEF model. As already explained, Commands normally get created by Policies. In our case, we also use the Commands to build the Universe graph in the observer `EOGObserver` (see Figure 4.7 on page 86).

EditorParts Package [ch.ethz.inf.sct.inference.visualizer.parts] The parts package contains all EditPart controller classes. See the description in the Controller subsection on page 86.

Policies Package [ch.ethz.inf.sct.inference.visualizer.policies] The policies are settled in this package. Although not all policies are really needed to fulfill the current use case of the Universe visualization, we implemented most of them. They are needed for an enhanced interaction with the graph (e.g., modification and re-inference of the Universe structure due to moving classes between Universes).

RI Package [ch.ethz.inf.sct.inference.visualizer.ri] Contains the Runtime Inference related classes. Currently, this is only the observer class `EOGObserver`.

SI Package [ch.ethz.inf.sct.inference.visualizer.si] Package for classes related to the GEF model build-up out of a Static Inference result.

Notes on the CVS structure and instructions about updating and deploying the plug-ins can be found in Appendix D and E.

Chapter 5.

Results and Conclusions

In this chapter we discuss the results of this thesis. Some examples that demonstrate the functionality of the tool are presented in Section 5.1.1 and 5.1.2. Possible future work that could improve the existing implementation is presented in Section 5.2. Section 5.3 completes this report with the conclusions.

5.1. Results

5.1.1. Combining Runtime and Static Inference

We use the same example as in Listing 2.4 in Section 2.3. The global preferences that were chosen for this example were quite sneaky, therefore we got such a bad inference result. The global preferences were chosen such that field got annotated with **readonly**, object creation with **rep**, and local variables with **peer**. It also introduced a bad cast because of this global preference settings. Nevertheless, considering the following example shows that, although the global preference settings are bad, the local preferences can correct that using the Combined Inference.

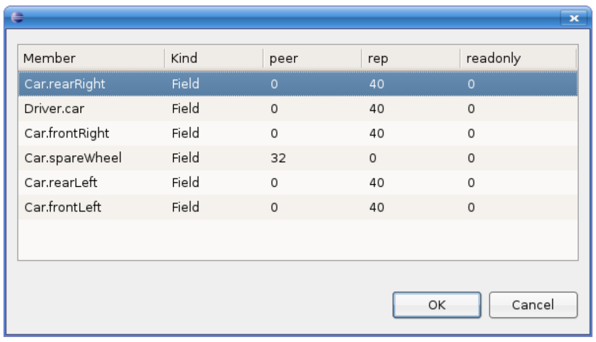
Listing 5.1: Results of using the Combined Inference.

```java
public class Car {
    /*@ rep @*/ Wheel frontLeft;
    /*@ rep @*/ Wheel frontRight;
    /*@ rep @*/ Wheel rearLeft;
    /*@ rep @*/ Wheel rearRight;
    /*@ peer @*/ Wheel spareWheel;

    public Car() {
        frontLeft = new /*@ rep @*/ Wheel();
        frontRight = new /*@ rep @*/ Wheel();
        rearLeft = new /*@ rep @*/ Wheel();
        rearRight = new /*@ rep @*/ Wheel();
        spareWheel = new /*@ peer @*/ Wheel();
    }
}

public class Driver {
    private /*@ rep @*/ Car car;

```



Member	Kind	peer	rep	readonly
Car.rearRight	Field	0	40	0
Driver.car	Field	0	40	0
Car.frontRight	Field	0	40	0
Car.spareWheel	Field	32	0	0
Car.rearLeft	Field	0	40	0
Car.frontLeft	Field	0	40	0

Figure 5.1.: The local preferences that were computed for the wheel example.

```
    public Driver() {
        car = new /*@ rep @*/ Car();
    }

    public void doJob() {
        /*@ rep @*/ Wheel sw;
        sw = car.spareWheel;
        sw.deflate();
    }

    public static void main(/*@ peer readonly @*/ String[] args) {
        /*@ peer @*/ Driver me = new /*@ peer @*/ Driver();
        me.doJob();
    }
}

public class Wheel {
    private float pressure;

    public void deflate() {
        pressure = 1.0f;
    }
}
```

The Runtime Inference inferred all normal wheels as **rep** and the spare wheel as **peer**. Therefore it sets in the local preferences quite high weights for the **rep** modifier of the wheels and **peer** for the spare wheel. Using all heuristics the local preferences were computed as shown in Figure 5.1. This forced the SAT solver to infer a **peer** annotation for the `spareWheel` and the other wheels with **rep**. Since the `Car` object and `Wheel sw` are in the same context they are both **rep** to the `Driver` object and can be safely deflated.

This example showed that the Static Inference alone is only as good as the preferences are set by the user (see the bad example in Listing 2.4 on page 26). Using the Combined Inference, bad global preferences can be repaired through the local preferences that are calculated using the Runtime Inference.

The Combined Inference is also suited to get rid of code coverage issues as discussed in Section 2.2. Listing 5.2 shows the same example as Listing 2.1.

Listing 5.2: No coverage issues with the Combined Inference.

```
public class Coverage {
    /*@ peer @*/ Coverage field1;
    /*@ peer @*/ Coverage field2;

    public void foo(boolean condition) {
        field1 = new /*@ peer @*/ Coverage();
        field2 = new /*@ peer @*/ Coverage();
        if (condition) {
            field1.makePeer(this);
        } else {
            field2.makePeer(this);
        }
    }

    public void doWrite() {
        // creates a write reference
    }

    public void makePeer(/*@ peer @*/ Coverage other) {
        other.doWrite();
    }

    public static void main(/*@ peer readonly @*/ String[] args) {
        /*@ peer @*/ Coverage c = new /*@ peer @*/ Coverage();
        c.foo(false);   // or c.foo(true) for other path
    }
}
```

Contrary to the Runtime Inference solution where one field got annotated with **rep**, both fields are annotated with **peer** using the Combined Inference. Table 5.1.1 shows the local preferences that were calculated using the heuristics "Class Scope Coverage Heuristic" and "Parameter Heuristic". The very low weights for the fields are effected by the coverage heuristic. Our coverage metric mesured a relatively low coverage value of 20%. Therefore, the weights for the `field1` and `field2` are also very low although the Runtime Inference inferred a **rep** annotation for `field1`. The annotation suggestion for the method parameter `Coverage other` from the Runtime Inference is supported by the parameter heuristic. That is why it suggests a relatively high preference for **peer**.

We have seen in this example that potentially wrong annotation suggestions caused by bad code coverage can be fixed with the coverage heuristic. The result of the Combined Inference is correct and corresponds to our expectations.

Member	Kind	peer	rep	readonly
Coverage.field2	Field	3	3	3
Coverage.field1	Field	3	3	3
Coverage.makePeer(Coverage):PARAM0	Parameter	34	3	3

Table 5.1.: The local preferences determined for the Coverage.

5.1.2. Eclipse Integration

The goal of this master thesis was to develop the Eclipse plug-ins for the Universe inference tools. We implemented a collection of 10 plug-ins that build the Universe Inference tools for Eclipse.

We conclude that the Eclipse plug-in infrastructure is a powerful framework that enables, after the time it takes to familiarize oneself with it, to develop plug-ins with quite powerful functionality. The problems we encountered were different integration issues. The design of the existing inference projects was not always consistent with the Eclipse architecture style. The separation of model, view, and controller in the command line tools was different than the Eclipse state-of-the-art MVC separation. The command line tools were designed to run in a single process with one main thread. Contrary, the Eclipse integration supports several concurrent worker threads and synchronisation with the SWT UI-thread. Further, we also had to integrate the logging facilities to Eclipse. Sometimes there were a lot of minor issues that led to code refactorings that need some time and are error-prone.

GEF turned out to be a very productive framework to create graphical applications in Eclipse. Offering a well structured and predefined architecture, we were quite amazed how fast one can develop with GEF. The whole Universe visualization was done in around three weeks. However, GEF has its drawbacks as well. It is not easy to find good documentation about it. Because the framework is still under heavy development, the source code is not yet well documented with Javadoc. Sometimes we had to look up the GEF example source code to understand how things work in GEF. We also noticed some conceptual issues with GEF. Connections are arranged in a special layer which is always the top layer. Therefore, z-buffering is not working properly (see Figure 5.2).

Due to time constraints we were not able to implement all desired functionality of the Universe visualizer, such as the interaction with the inferer. Nevertheless, we built the functionality of manipulating Universes and moving nodes from one Universe to another.

5.2. Future Work

Annotator

Semantic Checks: Perform semantic checks during the annotation phase. Report errors if an annotation file contains semantically illegal annotations.

XML: Use caching to speed up the lookup of annotations in the XML DOM. Probably also use XPath or XQuery instead of iterating through the

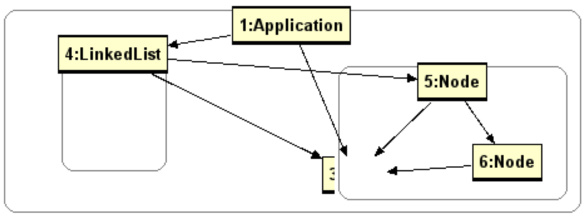


Figure 5.2.: The class instance 3:Node is covered behind a Universe. Nevertheless, the connections to the node are painted because of the missing z-buffering in GEF.

DOM.

Source Layout: Respect original source code layouts (whitespaces, indents, and linespaces).

Eclipse AST: Use Eclipse AST and the Eclipse refactoring API to insert Universe modifiers.

Runtime Inference

Tracing with JDI: It might be possible to get rid of the intermediate step of the tracing and directly build up the EOG while tracing events. This could be possible by re-implementing the tracing agent using the Java Debugging Interface (JDI).

Merge: Instead of merging multiple trace files in the EOG it would also be possible to merge multiple annotation XML files in the Static Inference.

Classloading Issues: We encountered different classloading issues with the abstract interpretation in Eclipse. While we use our own classloader implementation in Eclipse to load classes from a project directory, BCEL uses an internal classloader. This led to incompatibility issues.

JVM Crashes: We encountered the issue that the tracing agent crashed the JVM when autoboxing was used in the source code. Unfortunately, the issue is not deterministic. Sometimes the tracing works fine, sometimes not.

Static Inference

Heuristic: Improve and enhance heuristics. Conduct extensive case studies to get more empirical data and adjust the heuristics.

Pre-Annotations: We currently set constant weights for Universe modifiers in pre-annotated sources. It might be interesting to use heuristics and set dynamic weights for the pre-annotations.

Generics: Enhance the Static Inference in order to support Universe annotations for generics.

Preferences:	Extend the global and local preferences for static method annotations.
Bad Casts:	Extend the Static Inference in a way that bad cast can be eliminated by considering the runtime behavior in the EOG.

Visualizer

SI Visualizer:	Extend the Visualizer for the Static Inference in order to visualize the whole static structure of a program.
Interaction:	Extend the Visualizer so that interaction with the inferer model is possible. It would be nice to change Universe modifiers by moving objects between the Universes. Automatically re-infer annotations after modifications in the Universe graph.
Layout:	Improve the automatic graph layout. Prevent the crossing of references with objects.
Persistence:	There was an issue with reconstrucing the layout from persistent visualization files. After saving manually edited Universe layout, the layout was not reconstructed and the automatic graph layout was applied. Due to lack of time we were not able to fix this issue.
Connections:	Use curved connections with splines instead of polygone connections. Currently this is not supported with GEF, but announced as a feature of a future GEF release.
Export:	Export of the Universe graph to images (SVG, PDF, EPS).
Snap-to-G:	Enhance the visualizer with placement tools. Use *grids* for dragging and resizing confined to grid coordinates, or dragging and resizing snap to the rows and columns implied by existing objects in the diagram *graph*. Also, use user defined horizontal and vertical *guides*.

5.3. Conclusion

In this master thesis we presented an approach to combine Runtime and Static Inference and the integration of the inference tools into the Eclipse development environment. As result both tools are now fully integrated in Eclipse and can easily be used in the normal Java perspective together with the already existing JML plug-ins.

With the inference tools a programmer is now able to infer Universe types using a powerful user interface. Universe modifiers can be inferred by either using the Runtime Inference or the Static Inference alone or as a combination together. The inference settings can be configured with a few mouse clicks and do not need complicated editing of XML files. The inference results are presented in a dedicated view that enables easy interaction with the Java editor. The inference result can be reviewed, fixed, and maybe re-inferred in an easy manner. The Universe ownership modifier can easily be inserted into the Java source using the annotator plug-in.

The Inference Visualizer can be used to visualize the different steps of the Runtime Inference algorithm and the resulting Universe structure. It is also extendable to be used for the Static Inference as well. The visualization provides automatic layouting facilities and allows the programmer to edit and interact with the Universe graph.

Finally, all the plug-ins are packaged into Eclipse features which are deployable over update sites with the Eclipse Update Manager.

Bibliography

[1] JavaCC - Java Compiler Compiler. Available from https://javacc.dev.java.net/.

[2] JTB - Java Tree Builder. Website of UCLA Compilers Group. Available from http://compilers.cs.ucla.edu/jtb/.

[3] Fadi Aloul. PBS v2.1: Incremental pseudo-boolean backtrack search SAT solver and optimizer, 2003. Available from http://www.eecs.umich.edu/~faloul/Tools/pbs/,.

[4] Marco Bär. Practical runtime universe type inference. Master's thesis, ETH Zurich, May 2006.

[5] Paolo Bazzi. Integration of universe type system tools into eclipse. Semester thesis, ETH Zurich, September 2006.

[6] S. Bergman and E. L. Lozinskii. A fast algorithm for max-sat approximation. pages 424 – 431, 2003.

[7] Robert V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley Professional, 1st edition, 1999.

[8] Eric Clayberg and Dan Rubel. *Eclipse: Building Commercial-Quality Plug-ins*. Addison-Wesley Professional, 2nd edition, 2006.

[9] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. Multijava project. OOPSLA, 2000. Available from http://multijava.sourceforge.net/.

[10] W. Cohen, P. Ravikumar, and S. Fienberg. Secondstring project. Available from http://secondstring.sourceforge.net/.

[11] W. Dietl, S. Drossopoulou, and P. Müller. Generic Universe Types. In *Foundations and Developments of Object-Oriented Languages (FOOL/WOOD '07)*, January 2007.

[12] W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology (JOT)*, 4(8):5–32, October 2005.

[13] A. L. Baker G. T. Leavens and C. Ruby. Preliminary design of JML: A behavioral interface specification language for java, 2004. See http://www.jmlspecs.org/.

[14] Erich Gamma and Kent Beck. *Eclipse erweitern: Prinzipien, Patterns und Plug-Ins*. Addison-Wesley, München, 2004.

[15] Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Kiem-Phong Vo. A technique for drawing directed graphs. *Software Engineering*, 19(3):214–230, 1993.

[16] Joseph R. Horgan and Aditya P. Mathur. Software testing and reliability. pages 531–566, 1996.

[17] Agitar Software Inc. The open quality initiative, 2006. Available from http://www.agitar.com/openquality/initiative.html.

[18] Sun Microsystems Inc. Code conventions for the java(TM) programming language, April 1999. Available from http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html.

[19] Sun Microsystems Inc. JSR 202: Java class file specification update, 2006. Available from http://www.jcp.org/en/jsr/detail?id=202.

[20] Sun Microsystems Inc. What is the type checking verifier?, 2006. Available from https://jdk.dev.java.net/CTV/learn.html.

[21] M. A. Jaro. Probabilistic linkage of large public health data files. *Statistics in Medicine 14*.

[22] Nathalie Kellenberger. Static runtime inference. Master's thesis, ETH Zurich, 2005.

[23] Xavier Leroy. Java bytecode verification: algorithms and formalizations. *Journal of Automated Reasoning*, pages 235 – 269, 2003.

[24] Frank Lyner. Runtime universe type inference. Master's thesis, ETH Zurich, 2005.

[25] Manoel Marques. Plugging in a logging framework for eclipse plug-ins. *IBM Developer Works*, 2004.

[26] Marco Meyer. Interaction with ownership graphs. Semester thesis, ETH Zurich, January 2006.

[27] Alvaro E. Monge and Charles Elkan. The field matching problem: Algorithms and applications. In *Knowledge Discovery and Data Mining*, pages 267–270, 1996.

[28] P. Müller and A. Poetzsch-Heffter. Universes: A type system for alias and dependency control. Technical Report 279, Fernuniversität Hagen, 2001.

[29] Matthias Niklaus. Static universe type inference using a SAT-solver. Master's thesis, ETH Zurich, May 2006.

[30] Apache XML Project. Java XMLBeans, 2003. Available from http://xmlbeans.apache.org.

[31] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.

[32] Tagawa S. Sugiyama, K. and M. Toda. Methods for visual understanding of hierarchical system structures. *IEEE Trans. on Sys. Man and Cyb*, pages 109–125, 1981.

[33] William E. Winkler. The state of record linkage and current research problems. *Statistics of Income Division, Internal Revenue Service Publication R99/04*, 1999.

Appendix A.

Generation of Parser

Because further work might has to enhance the annotator grammar file and generate a new parser, we append our ANT script to ease the tasks of a future student:

```
<project name="AnnotationTool␣Build␣Parser" default="" basedir=".">

    <target name="jtb" description="Generate␣the␣tree␣builder␣grammer␣using␣JTB.
        ">
        <java classname="EDU.purdue.jtb.JTB">
        <arg line="−p␣EDU.purdue.jtb␣−jd␣−f␣−tk␣−printer␣Java1.5.jj" />
        <classpath>
            <pathelement location="lib/jtb132mod.jar"/>
        </classpath>
        </java>
        <copy todir="src/EDU/purdue/jtb/visitor">
            <fileset dir="visitor"/>
        </copy>
        <copy todir="src/EDU/purdue/jtb/syntaxtree">
            <fileset dir="syntaxtree"/>
        </copy>
        <delete dir="visitor"/>
        <delete dir="syntaxtree"/>
    </target>

    <target name="javacc" description="Generate␣the␣parser␣using␣JavaCC.">
        <mkdir dir="src/EDU/purdue/jtb/parser" />
        <javacc
           target="jtb.out.jj"
           outputdirectory="src/EDU/purdue/jtb/parser"
           javacchome="D:/Programme/javacc−4.0"
        />
    </target>

    <target name="compileparser" description="Compiles␣the␣generated␣parser">
        <javac sourcepath="" srcdir="src" destdir="bin" >
            <include name="EDU/purdue/jtb/∗.java"/>
        </javac>
    </target>
```

```
  <target name="correct_comment" description="Corrects␣the␣code␣and␣javadoc␣
      comments">
      <replaceregexp
         match="&quot;\*/&quot;"
         replace="&quot;(star)/&quot;" >
          <fileset dir="src/EDU/purdue/jtb" includes="**/*.java" />
      </replaceregexp>
  </target>

  <target name="all" depends="jtb,␣javacc,␣correct_comment,␣compileparser">
  </target>

  <target name="clean" description="Cleans␣all␣the␣generated␣files">
      <delete dir="bin/EDU" />
      <delete dir="src/EDU/purdue/jtb/syntaxtree" />
      <delete dir="src/EDU/purdue/jtb/visitor" />
      <delete>
          <fileset dir="src/EDU/purdue/jtb/parser">
              <include name="JTBParser.java"/>
              <include name="JTBParserTokenManager.java"/>
              <include name="JTBParserConstants.java"/>
              <include name="*CharStream*.java"/>
              <include name="TokenMgrError.java"/>
              <include name="ParseException.java"/>
          </fileset>
      </delete>
      <delete file="jtb.out.jj" />
  </target>

</project>
```

Appendix B.

Generation of XML Binding Classes

The accessing of the XML files is done by binding the XML to Java types. To compile the XML schema to Java classes, you have to perform the task described in the ANT script below. XMLBeans needs the JAR libraries xbean.jar and jsr173_api.jar. Both are in the CVS or can be downloaded from [30].

```
<project name="XMLBean" default="" basedir=".">

    <taskdef name="xmlbean" classname="org.apache.xmlbeans.impl.tool.XMLBean"
        classpath="lib/xbean.jar"/>

    <target name="compileTypes" description="Compile␣the␣XML␣Schema">
        <xmlbean classgendir="src" classpath="${classpath}"
            destfile ="xmltypes.jar"  failonerror ="true" javasource="1.5" >
             <fileset dir="src" excludes="**/*.xsd"/>
             <fileset dir="schemas" includes="**/annotations.xsd"/>
        </xmlbean>
    </target>
</project>
```

Appendix C.
Annotation XML Schema

C.1. Indexing

The indexing of parameters and assignments was not clear to me and not properly documented. Therefore, we try to describe the indexing and outline some problems:

The `AddCast` element in the annotation XML can have the position type `assignment`, `method_call`, `array_initialyzer`, and `return_stmt`.

Pos Type	Pos	Description	Example
assignment	-1	Add cast to target	`((peer C)x.y.z).field = a;`
	0	Add cast to the expression on the right hand side	`x.field = (rep C)a;`
	>0	not valid	
method_call	-1	Add cast to target	`((peer C)x.y.z).foo();`
	≥0	Add cast to the argument at this position	`x.foo((peer C)p);`
array_initializer	-1	not valid	
	≥0	Array entry at this position	`C[] = new C[] { (peer C)e1, e2 };`
return_stmt	-1	not valid	
	0	Add cast to the expression	`return (peer C)obj;`
	≥0	not valid	

Notice the case where a cast has to be added to `y = foo()`. Not to cast the return value of the call, but the target, one has to rewrite the expression to `y = ((peer C)this).foo()`. Semantically, this does not make any sense because `this` is always a **peer** reference. Casts only have to be inserted in the case of a downcast of a **readonly** target that is calling a non-**pure** method. This would require a semantic check. Semantic checks are not supported in the current implementation, therefore the annotator does not perform the transformation described above.

The XSD schema allows to add different casts for the elements of an array. Similar to method parameters, a position attribute identifies the element at the associated position. So, it is possible to add semantically incorrect casts since the array components all have to be of the same type. Something like `peer peer C[] cArray = new C[] { new rep C(), new peer C() }` is wrong.

We do not perform any semantic checks with the annotation tool. We assume that annotations in the XML are semantically correct and simply insert it into the Java sources.

C.2. Annotation XML Schema [annotations.xsd]

```
<?xml version="1.0" encoding="UTF−8"?>
<!−−
Schema for annotation files that specify Universe annotations that
should be added to existing sources.

Author: WMD

$Id: annotations.xsd,v 1.6 2007/03/11 14:40:00 anfuerer Exp $

Revision History:
− (anfuerer 10.03.07)
  Wrong schema was checked in to CVS. Corrected that.
− (anfuerer 21.09.06)
  Made schema of RI and SI consistent
− (anfuerer 23.09.06)
  Some minor changes (typo, formatting)
− (anfuerer 10.12.06)
  Updated description
− (anfuerer 29.01.07)
  Updated description comment for index of addcast.
− (anfuerer 08.03.07)
  Minor formatting changes
−−>
<xsd:schema xmlns:po="http://sct.inf.ethz.ch/annotations" xmlns:xsd="http://www.w3.
    org/2001/XMLSchema" targetNamespace="http://sct.inf.ethz.ch/annotations">


<!−−XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Some additional types to automatically check the input.
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX −−>

<!−−
The modifiers that are valid for simple reference types.
−−>
<xsd:simpleType name="SimpleUniverseModifier">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="implicit_peer"/>
    <xsd:enumeration value="peer"/>
    <xsd:enumeration value="rep"/>
    <xsd:enumeration value="readonly"/>
  </xsd:restriction>
</xsd:simpleType>

<!−−
```

```
The modifiers that are valid for types, including arrays.
-->
<xsd:simpleType name="UniverseModifier">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="implicit_peer"/>
    <xsd:enumeration value="peer"/>
    <xsd:enumeration value="rep"/>
    <xsd:enumeration value="readonly"/>

    <xsd:enumeration value="peer␣peer"/>
    <xsd:enumeration value="peer␣readonly"/>
    <xsd:enumeration value="rep␣peer"/>
    <xsd:enumeration value="rep␣readonly"/>
    <xsd:enumeration value="readonly␣peer"/>
    <xsd:enumeration value="readonly␣readonly"/>
  </xsd:restriction>
</xsd:simpleType>


<!--
The modifiers that are valid for methods.
-->
<xsd:simpleType name="UniverseMethodModifier">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value=""/>
    <xsd:enumeration value="pure"/>
  </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="CastPositionType">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="assignment"/>
    <xsd:enumeration value="method_call"/>
    <xsd:enumeration value="array_initialyzer"/>
    <xsd:enumeration value="return_stmt"/>
  </xsd:restriction>
</xsd:simpleType>


<!--
What target should be modified?
-->
<xsd:simpleType name="ToolTarget">
  <xsd:restriction base="xsd:string">
    <!-- Modify the original Java sources -->
    <xsd:enumeration value="java"/>
    <!-- Create JML specification files -->
    <xsd:enumeration value="jml"/>
  </xsd:restriction>
</xsd:simpleType>


<!--
With what style should the annotations be inserted?
-->
<xsd:simpleType name="ToolStyle">
  <xsd:restriction base="xsd:string">
    <!-- As standard type annotations, e.g. "peer␣T" -->
    <xsd:enumeration value="types"/>

    <!-- Within JML comments, e.g. "/*@␣peer␣T␣@*/" -->
    <xsd:enumeration value="jml"/>

    <!-- As escaped JML comments, e.g. "/*@␣\peer␣T␣@*/" -->
    <xsd:enumeration value="oldjml"/>
  </xsd:restriction>
</xsd:simpleType>


<!--XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
The elements of our schema.
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX-->

<!--
The top-level element consisting of one header element and
at least one class element.
-->
<xsd:element name="annotations">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element maxOccurs="1" minOccurs="1" ref="po:head"/>
      <xsd:element maxOccurs="unbounded" minOccurs="1" ref="po:class"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>


<!--
Some additional information at the beginning.
Should be overridable on the command line.
-->
<xsd:element name="head">
  <xsd:complexType>
```

```
    <xsd:sequence>
      <!-- Should we create a ".jml" specification or embed the
           annotations in existing ".java" files ? -->
      <xsd:element name="target" type="po:ToolTarget"/>

      <!-- What style of Universe annotations should we use? -->
      <xsd:element name="style" type="po:ToolStyle"/>

      <!-- Maybe the source of the annotations. -->
      <xsd:element name="comment" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>


<!--
The annotations for one class.
-->
<xsd:element name="class">
  <xsd:complexType>
    <xsd:sequence>
      <!-- Annotations for the fields of the class. -->
      <xsd:element maxOccurs="unbounded" minOccurs="0" ref="po:field"/>

      <!-- Annotations for the methods of the class. -->
      <xsd:element maxOccurs="unbounded" minOccurs="0" ref="po:method"/>

      <!-- Annotations for the object initializers . -->
      <xsd:element maxOccurs="unbounded" minOccurs="0" name="object_init" type="
          po:object_class_init"/>

      <!-- Annotations for the class initializers . -->
      <xsd:element maxOccurs="unbounded" minOccurs="0" name="class_init" type="
          po:object_class_init"/>
    </xsd:sequence>

    <!-- The fully qualified name of the class. -->
    <xsd:attribute name="name" type="xsd:string" use="required"/>

    <!-- Optionally, the relative path to the source file . -->
    <xsd:attribute name="file" type="xsd:string"/>

  </xsd:complexType>
</xsd:element>


<!--
```

```
The annotation for a field .
-->
<xsd:element name="field">
  <xsd:complexType>
    <xsd:sequence>
      <!-- The annotations for the field initializer . -->
      <xsd:element maxOccurs="1" minOccurs="0" ref="po:field_init"/>
    </xsd:sequence>

    <!-- The name of the field. -->
    <xsd:attribute name="name" type="xsd:string" use="required"/>

    <!-- The Java type of the field. -->
    <xsd:attribute name="type" type="xsd:string" use="required"/>

    <!-- Optionally, the source line of the declaration .
         Would this really help a tool to insert the annotation?
         What if there is more than one declaration per line ?
    -->
    <xsd:attribute name="line" type="xsd:int"/>

    <!-- One of the Universe modifiers. -->
    <xsd:attribute default="implicit_peer" name="modifier" type="po:UniverseModifier"/
        >

  </xsd:complexType>
</xsd:element>


<!--
The annotations for a method or constructor.
-->
<xsd:element name="method">
  <xsd:complexType>
    <xsd:sequence>
      <!-- The annotation for the return type. -->
      <xsd:element maxOccurs="1" minOccurs="0" ref="po:return"/>

      <!-- The annotations for the parameter types. -->
      <xsd:element maxOccurs="unbounded" minOccurs="0" ref="po:parameter"/>

      <!-- The annotations for the local variables. -->
      <xsd:element maxOccurs="unbounded" minOccurs="0" ref="po:local"/>

      <!-- The annotations for object creations in this method. -->
      <xsd:element maxOccurs="unbounded" minOccurs="0" ref="po:new"/>
      <!-- The annotations for casts in this method. -->
      <xsd:element maxOccurs="unbounded" minOccurs="0" ref="po:cast"/>

      <!-- The annotations for casts in this method. -->
      <xsd:element maxOccurs="unbounded" minOccurs="0" ref="po:addcast"/>

      <!-- The annotations for static calls in this method. -->
      <xsd:element maxOccurs="unbounded" minOccurs="0" ref="po:static_call"/>
    </xsd:sequence>

    <!-- The name of the method. -->
    <xsd:attribute name="name" type="xsd:string" use="required"/>

    <!-- The signature of the method. Multiple methods can have the
         same name, the signature resolves the overloading. -->
    <xsd:attribute name="signature" type="xsd:string" use="required"/>

    <!-- Optionally, the source line of the declaration.
         Would this really help a tool to insert the annotation?
         What if there is more than one declaration per line?
    -->
    <xsd:attribute name="line" type="xsd:int"/>

    <!-- Modifiers that should be added to the method.
         At the moment there is only "pure" or "". -->
    <xsd:attribute default="" name="modifier" type="po:UniverseMethodModifier"/>
  </xsd:complexType>
</xsd:element>


<!--
The annotations for a field initialzier.
-->
<xsd:element name="field_init">
  <xsd:complexType>
    <xsd:sequence>
      <!-- The annotations for object creations in this initializer. -->
      <xsd:element maxOccurs="unbounded" minOccurs="0" ref="po:new"/>

      <!-- The annotations for casts in this initializer. -->
      <xsd:element maxOccurs="unbounded" minOccurs="0" ref="po:cast"/>

      <!-- The annotations for casts in this initializer. -->
      <xsd:element maxOccurs="unbounded" minOccurs="0" ref="po:addcast"/>

      <!-- The annotations for static calls in this initializer. -->
      <xsd:element maxOccurs="unbounded" minOccurs="0" ref="po:static_call"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>


<!--
The annotations for an object or class initializer .
Careful: all the initializer blocks are merged into one of each kind
for execution.
So if the annotation information comes from the runtime inference tool,
the indices might be larger than expected from one initializer alone.
-->
<xsd:complexType name="object_class_init">
    <xsd:sequence>
      <!-- The annotations for the local variables. -->
      <xsd:element maxOccurs="unbounded" minOccurs="0" ref="po:local"/>

      <!-- The annotations for object creations in this method. -->
      <xsd:element maxOccurs="unbounded" minOccurs="0" ref="po:new"/>

      <!-- The annotations for casts in this method. -->
      <xsd:element maxOccurs="unbounded" minOccurs="0" ref="po:cast"/>

      <!-- The annotations for casts in this method. -->
      <xsd:element maxOccurs="unbounded" minOccurs="0" ref="po:addcast"/>

      <!-- The annotations for static calls in this method. -->
      <xsd:element maxOccurs="unbounded" minOccurs="0" ref="po:static_call"/>
    </xsd:sequence>

    <!-- The index of the initializer within the class,
          starting from zero.
    -->
    <xsd:attribute name="index" type="xsd:unsignedInt" use="required"/>

    <!-- Optionally, the source line of the opening "{". -->
    <xsd:attribute name="line" type="xsd:int"/>

    <!-- Modifiers that should be added to the method.
          At the moment there is only "pure" or "".
          Not supported yet, but might come...
          <xsd:attribute name="modifier" type="UniverseMethodModifier" default=""/>
    -->
</xsd:complexType>


<!--
```

```
The annotation for the return type.
-->
<xsd:element name="return">
  <xsd:complexType>
    <!-- The Java type of the return value. -->
    <xsd:attribute name="type" type="xsd:string" use="required"/>

    <!-- Optionally, the source line of the declaration.
         Would this really help a tool to insert the annotation?
         What if there is more than one declaration per line?
    -->
    <xsd:attribute name="line" type="xsd:int"/>

    <!-- One of the Universe modifiers. -->
    <xsd:attribute default="implicit_peer" name="modifier" type="po:UniverseModifier"/
        >
  </xsd:complexType>
</xsd:element>


<!--
The annotation for a parameter.
-->
<xsd:element name="parameter">
  <xsd:complexType>
    <!-- The index of the parameter, starting from zero.
         Might be the only thing available. -->
    <xsd:attribute name="index" type="xsd:unsignedInt" use="required"/>

    <!-- The Java type of the parameter. -->
    <xsd:attribute name="type" type="xsd:string" use="required"/>

    <!-- The name of the parameter, if available.
         Otherwise "param" + index is used as name if needed. -->
    <xsd:attribute name="name" type="xsd:string"/>

    <!-- Optionally, the source line of the declaration.
         Would this really help a tool to insert the annotation?
         What if there is more than one declaration per line?
    -->
    <xsd:attribute name="line" type="xsd:int"/>

    <!-- One of the Universe modifiers. -->
    <xsd:attribute default="implicit_peer" name="modifier" type="po:UniverseModifier"/
        >
  </xsd:complexType>
</xsd:element>
```

```
<!--
The annotation for a local variable.
-->
<xsd:element name="local">
  <xsd:complexType>
    <!-- The index of the local variable, starting from zero.
         Might be the only thing available. -->
    <xsd:attribute name="index" type="xsd:unsignedInt" use="required"/>

    <!-- The Java type of the local variable. -->
    <xsd:attribute name="type" type="xsd:string" use="required"/>

    <!-- The name of the local variable, if available.
         Otherwise "local" + index is used as name if needed. -->
    <xsd:attribute name="name" type="xsd:string"/>

    <!-- Optionally, the source line of the declaration.
         Would this really help a tool to insert the annotation?
         What if there is more than one declaration per line?
    -->
    <xsd:attribute name="line" type="xsd:int"/>

    <!-- One of the Universe modifiers. -->
    <xsd:attribute default="implicit_peer" name="modifier" type="po:UniverseModifier"/
        >
  </xsd:complexType>
</xsd:element>


<!--
The annotation for an object creation.
The existing new expressions in a method are indexed, starting from zero.
-->
<xsd:element name="new">
  <xsd:complexType>
    <!-- The index of the new, starting from zero. -->
    <xsd:attribute name="index" type="xsd:unsignedInt" use="required"/>

    <!-- The Java type of the new. -->
    <xsd:attribute name="type" type="xsd:string" use="required"/>

    <!-- Optionally, the source line of the new.
         Would this really help a tool to insert the annotation?
         What if there is more than one new per line?
    -->
    <xsd:attribute name="line" type="xsd:int"/>

    <!-- One of the Universe modifiers. -->
    <xsd:attribute default="implicit_peer" name="modifier" type="po:UniverseModifier"/
        >

  </xsd:complexType>
</xsd:element>


<!--
The annotation for a cast.
At the moment this is very limited.
The existing casts in a method are indexed, starting from zero.
No new casts can be introduced.
How could we exactly say where a new cast should be inserted??
-->
<xsd:element name="cast">
  <xsd:complexType>
    <!-- The index of the cast, starting from zero. -->
    <xsd:attribute name="index" type="xsd:unsignedInt" use="required"/>

    <!-- The Java type of the cast. -->
    <xsd:attribute name="type" type="xsd:string" use="required"/>

    <!-- Optionally, the source line of the cast.
          Would this really help a tool to insert the annotation?
          What if there is more than one cast per line?
    -->
    <xsd:attribute name="line" type="xsd:int"/>

    <!-- One of the Universe modifiers. -->
    <xsd:attribute default="implicit_peer" name="modifier" type="po:UniverseModifier"/
        >
  </xsd:complexType>
</xsd:element>


<!--
The annotation for an additional cast.
At the moment we only support static type inference tools.
-->
<xsd:element name="addcast">
  <xsd:complexType>

    <!-- The Java type of the cast. -->
    <xsd:attribute name="type" type="xsd:string" use="required"/>
    <!-- Optionally, the source line of the cast.
          Would this really help a tool to insert the annotation?
          What if there is more than one cast per line?
    -->
    <xsd:attribute name="line" type="xsd:int" use="optional"/>

    <!-- One of the Universe modifiers. -->
    <xsd:attribute name="modifier" type="po:UniverseModifier" use="required"/>

    <!-- One of the possible postitions to insert casts method, assignment or
          array initializer -->
    <xsd:attribute name="position_type" type="po:CastPositionType" use="required"/>

    <!-- The index of the position. starting from zero -->
    <xsd:attribute name="index" type="xsd:unsignedInt" use="required"/>

    <!-- The position in the position type. -1 means target. A value greater or equal 0
          means the parameter at this position for position type method_call. Notice that
          parameters are numbered starting from 0 until (n-1) where n is the number of
          parameters.
          For an assignment 0 is the expression to assign (the right hand side)
           -->
    <xsd:attribute name="position" type="xsd:int" use="required"/>

  </xsd:complexType>
</xsd:element>


<!--
The annotation for a static method call.
The existing static method calls in a method are indexed, starting from zero.
One index for each target class is maintained. The target class needs to be specified
in the static method call.
-->
<xsd:element name="static_call">
  <xsd:complexType>
    <!-- The index of the static call, starting from zero. -->
    <xsd:attribute name="index" type="xsd:unsignedInt" use="required"/>

    <!-- The Java type of the call. -->
    <xsd:attribute name="type" type="xsd:string" use="required"/>

    <!-- Optionally, the source line of the call.
          Would this really help a tool to insert the annotation?
          What if there is more than one new per line?
    -->
    <xsd:attribute name="line" type="xsd:int"/>

    <!-- One of the simple Universe modifiers, because static calls are
         not possible on array types.
    -->
    <xsd:attribute default="implicit_peer" name="modifier" type="
        po:SimpleUniverseModifier"/>
  </xsd:complexType>
</xsd:element>

</xsd:schema>
```

Appendix D.
CVS Structure

All Eclipse projects related with this thesis can be found in the CVS on host waldorf.inf.ethz.ch in the module `dietlw/projects/inference-pack/`.

The structure is as follows:

Directory	**Project**
annotation-tool-plugin	The annotator plug-in that parses Java sources and inserts Universe modifiers.
annotation-xml-plugin	XML annotation file plug-in. Contains the generated EMF model code.
annotation-xml-plugin.edit	XML annotation file plug-in. Contains the generated EMF code providing the model edit capabilities.
annotation-xml-plugin.editor	XML annotation file plug-in. Contains the generated EMF code for the editor UI.
inference-combined-plugin	Combined Inference plug-in project.
inference-common-plugin	Inference Common plug-in. Contains code that is shared by all inference Eclipse plug-ins.
inference-common-resources	Common Inference resources plug-in. Contains common libraries and code of all inference projects (also the resources projects).
inference-feature	Feature project for the deployment of the inference plug-ins.
inference-update-site	Update site project for the deployment with an Eclipse update site.
inference-visualizer-2	The Inference Visualizer plug-in.
jml-plugin	The JML plug-in.
jml-resources	The JML resources plug-in.
logging-plugin	The logging plug-in used for the log4j integration in Eclipse.
runtime-inference-config	The XML configuration file editor plug-in. Contains the generated EMF model code.
runtime-inference-config.edit	The XML configuration file editor plug-in. Contains the generated EMF code providing the model edit capabilities.
runtime-inference-config.editor	The XML configuration file editor plug-in. Contains the generated EMF code for the editor UI.

runtime-inference-plugin	The Runtime Inference Eclipse plug-in.
runtime-inference-resources	The Runtime Inference resources plug-in.
static-inference-plugin	The Static Inference Eclipse plug-in.
static-inference-resources	The Static Inference resources plug-in.

Appendix E. Plug-in Deployment

If you want to update a plug-in (e.g., perform bug-fix, update JML libraries, etc.) the following steps are necessary to deploy the plug-in:

- Check out latest CVS version of the corresponding plug-in.
- Replace libraries or other files with the new version.
- Open the file `MANIFEST.MF` and increase the plug-in version number.
- Use the `build_plugin.xml` to build a new version of the plug-in.
- Copy the new plug-in file from the `/dist` folder into the Eclipse `/plugin` folder and test it.
- Check out the latest CVS version of the feature project containing the plug-in to update.
- Increase the version number of the feature.
- Check out the latest CVS version of the update site project.
- Open the file `site.xml`, synchronize the changed feature and rebuild the feature containing the plug-in to update.
- Copy the new feature and plug-in files together with the changed `site.xml` file to the Web server providing the update site.

JAR libraries are settled in the resource projects. Updating the plug-ins with a new JML release only requests to replace the JML and MJ JARs in the JML resource plug-in.

Appendix F.

Example of the Jaro-Winkler Algorithm

F.1. Jaro Algorithm

The Jaro algorithm [21] is described as follows:

Given strings $s = a_1 \dots a_K$ and $t = b_1 \dots b_L$

Define a character a_i in s to be common with t.

There is a $b_j = a_i$ in t such that $i - H \leqslant j \leqslant i + H$, where $H = \frac{min(|s|,|t|}{2}$

Let $s' = a'_1 \dots a'_K$, be the characters in s which are *common with* t (they appear in the same order) and let $t' = b'_1 \dots a'_L$ be analogous.

Now define a *transposition for* s', t' to be a position i such that $a'_i \neq b'_i$.

Let $T_{s',t'}$ be half the number of transpositions for s' and t'. The Jaro similarity metric for s and t is

$$Jaro(s,t) = \frac{1}{3}\left(\frac{|s'|}{|s|} + \frac{|t'|}{|t|} + \frac{|s'| - T_{s',t'}}{|s'|}\right)$$

F.2. Example

Let $s = AAAAAAAA$
and $t = AAAABBBB$

$s' = AAAA$ and $t' = BBBB$

$$T_{s,t} = \frac{0}{2} = 0$$

$$Jaro(s,t) = \frac{1}{3}\left(\tfrac{4}{8} + \tfrac{4}{8} + \tfrac{4-0}{4}\right) = \tfrac{2}{3}$$

F.3. Jaro-Winkler Algorithm

A variant of this due to Winkler [33] also uses the length P of the *longest common prefix* of s and t.

Letting $P' = max(P, 4)$ we define

$$JaroWinkler(s,t) = Jaro(s,t) + \frac{P'}{10}\left(1 - Jaro(s,t)\right)$$

F.4. Example

$P = 4$ and $P' = max(P, 4) = 4$

$$JaroWinkler(s,t) = \frac{2}{3} + \frac{4}{10}\left(1 - \frac{2}{3}\right) = 0,8$$