Interaction with Ownership Graphs

Marco Meyer

Semester Project Report

Software Component Technology Group
Department of Computer Science
ETH Zurich

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

WS 05/06

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 is used to structure the object store and poses some rules to references between such objects. Java programs can be annotated with these rules called Universe Type annotations to control access between components. Research in this field has been done with the goal to infer Universe Types automatically. This report describes two projects supporting this work. First, we developed a tool to insert the obtained Universe Type annotations into java source code. The second project is a visualization tool for the algorithm of the Runtime Universe Type Inference project.

Contents

Chapter 1

Introduction

1.1 Overview

In object-oriented programming, references (especially reference passing) pose some problems in the context of proving correctness. Every object may have a reference to any other object, which complicates the program verification.

The Ownership model provides a way to structure and restrict the objects and their references. With the Universe Type System [1, 2], the programmer can express these ownership rules. This allows also the static checking without a large overhead.

Previous research has been done with the focus on computing these ownership rules automatically. This semester thesis supports an earlier project by contributing two tools.

1.2 Previous Work

In order to determine these Universe Types automatically, work has been done by static and runtime analysis of code. The Static Universe Type Inference [3] tool was developed by Nathalie Kellenberger as her master thesis. It obtains information by static code analysis.

Frank Lyner developed the Runtime Universe Type Inference [4] project, which is closely linked to the work of this semester thesis. In his master thesis, Frank Lyner found and implemented an algorithm to analyze a programs behaviour at runtime. He then receives information about the Universe Types out of such a runtime analysis. At the moment, Frank Lyners work is continued by Marco Baer in his master thesis Practical Runtime Universe Type Inference [5]. Amongst other things, the handling of arrays and static methods will be included.

1.3 Runtime Universe Type Inference

In this section, some information on the Runtime Universe Type Inference project is provided in order to get familiar with the points where the Runtime Type Inferer and the work of this semester thesis are connected. It is not intended to give a comprehensive summary. For more information on the Runtime Type Inferer see [4].

The Runtime Universe Type Inference project developed and implemented an algorithm that observes the runtime behaviour of a regular Java program. On the basis of this analysis Universe Type annotations are computed. These annotations are then saved in an XML format, corresponding to the XML Schema defined by the file annotations.xsd shown in Appendix A. Such an XML annotation file (per default called annotations.xml) contains not only the annotations but also all the information needed to insert the annotations into the original java source files. This XML file will be used by the Annotation Tool which is the first project of this semester thesis.

As a second interface to this semester thesis, the Runtime Type Inferer program provides an observer interface by which the Type Inference algorithm can be observed. To be exact, the interface lets us observe all the changes to the Extended Object Graph, which is built up by the Runtime Type Inferer. The Extended Object Graph is a snapshot of the heap at a specific point in time. This interface is used by the second project of this semester thesis, the Runtime Visualizer, to create a graphical representation of the Extended Object Graph. For more information on the observer interface, we refer to section 3.3 of [4].

1.4 Goal

The goal of this semester thesis is to support the research in the field of the Universe Type System with two projects (see Figure 1.1 for an overview). The first one, called Annotation Tool or Annotator, is a tool to annotate Java source files with Universe Type annotations. These annotations might for example be provided by one of the Universe Type Inference tools mentioned earlier (see 1.2).

The second project, named Runtime Visualizer, wants to visualize the process of finding the annotation information like it is done by the Runtime Universe Type Inference tool. Therefore, it creates a graphical representation of the Extended Object Graph, which then can be displayed at each step of its creation process. Such a graphical representation makes it easy to follow the creation and processing steps of the Extended Object Graph. In a further step, one might be able to interactively influence the inference process by doing modifications on the Extended Object Graph. This could be done in an intuitive way on such a graphical representation of the Extended Object Graph.

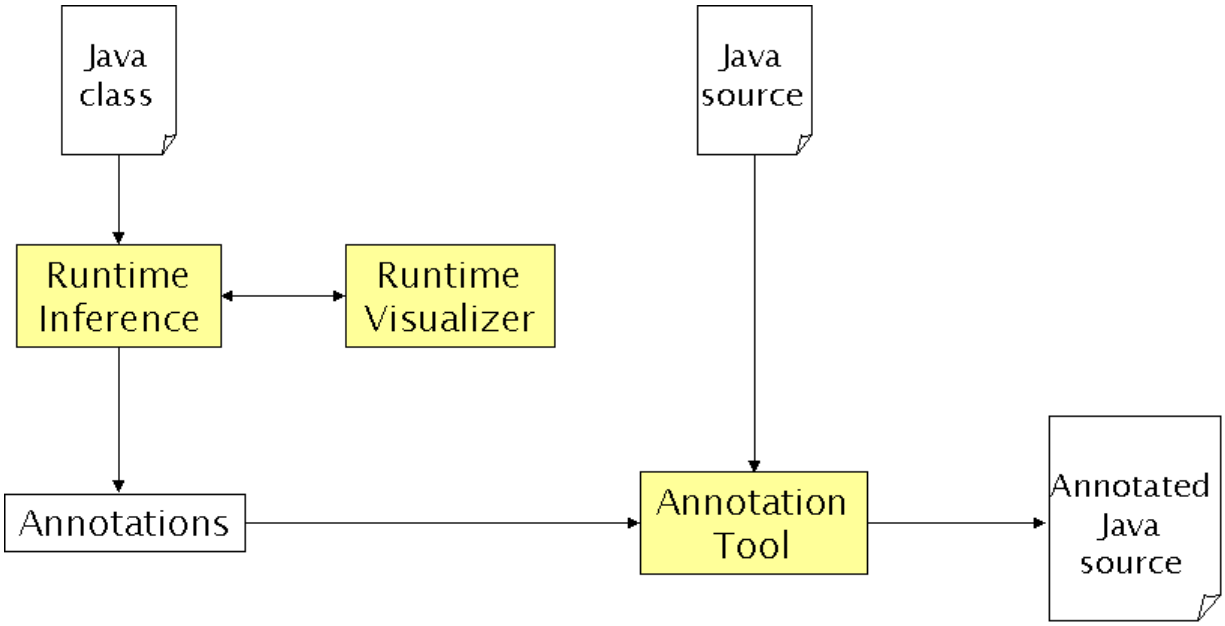

Figure 1.1: Overview

1.5 Outline

Since the two developed projects, the Annotation Tool and the Runtime Visualizer, are independent from each other, they are discussed in separate chapters. In chapter 2 the Annotation Tool is documented while the Runtime Visualizer is discussed in chapter 3. They are followed by the Bibliography and Appendix A.

Chapter 2

Annotation Tool

2.1 Motivation

The Runtime Type Inferer returns the gained knowledge about the Universe Type annotations as an XML file. This provides us with all the information that we need, to know to which method, type, etc. each annotation belongs to. But the annotations are still separated and not embedded in the Java program. What remains to be done now, is to insert the annotations into the java sources. In order to do this we need some sort of source code analyzer. It is obvious that a source code parser will do the job.

2.2 Scope

The Annotation Tool has to insert the Universe Type annotations found by the Runtime Type Inferer into the java source code at the proper locations. The Annotation Tool is intended to be used mainly with the output of the Type Inferer or some other tool which automatically generates the XML annotation information. Therefore it is not part of this project to do any checking of the annotation information. We can assume that the annotation information is consistent and correct (meaning: corresponds to the provided java source files).

2.3 Evaluation

For the task of inserting annotations into a java source file, we need some kind of source code parser. Since Java is a widely-used language, there exist a large number of Java parsers (e.g. [6, 7, 8, 9]). Although almost any of them would be sufficient to do the work, some are more convenient than others. First of all, it should be possible to insert the annotations during one

single parser invocation. We do not want to run the parser several times per source file. We do not intend to perform striking changes on the structure of the program, in fact, we even do not need to modify it at all. What we are basically interested in is just an output of the same java source file supplemented with the annotations. So we do not need complex functionality, but want to keep it as simple as possible.

Furthermore a desired feature of such a parser would be a Java 1.5 grammar. It would be impractical if our program cannot annotate Java 1.5 code. Therefore, parsers with Java 1.5 grammars are preferred.

2.4 Decision

After examining several products, the Java Tree Builder(JTB)[9] in combination with the Java Compiler Compiler (JavaCC)[10] was chosen. The JavaCC is a parser generator for use with Java applications. A parser generator reads a grammar specification and creates a Java program that can recognize matches to this grammar. The JTB is a syntax tree builder to be used with the JavaCC parser generator. It takes a plain JavaCC grammar file as input and automatically generates a set of syntax tree classes based on the productions in the grammar. These classes implement a visitor pattern, which can be used to explore or even modify the syntax tree. The JTB also generates a JavaCC grammar with the proper annotations to build the syntax tree during parsing. Both the JavaCC and the JTB are themselves written in Java. As we use the Java Tree Builder with version 1.3.2, at least Java 1.5 is required.

The usage of the Java Tree Builder is e.g. as follows:

```
java EDU.purdue.jtb.JTB -p EDU.purdue.jtb -jd -f -tk -printer jtbgram.jj
```

The usage of the Java Compiler Compiler is as follows:

```
javacc jtb.out.jj
```

with:

-p specifies the package of the generated parser

-tk generates the special tokens into the syntax tree

jtbgram.jj Java 1.5 grammar provided by the Java Tree Builder

jtb.out.jj Java 1.5 grammar supplemented with code for syntax tree generation. This file is generated by invocation of the JTB like shown above.

Basically, we wouldn't need to build our own parser. Since we need a standard Java 1.5 parser, we could use the one provided by the distribution (`jtb132.jar`). But in that version all comments are discarded. They are parsed as special tokens which per default are skipped and not inserted in the syntax tree. We do not want to leave the comments out because we need them in the output of our Annotation Tool. Therefore, we have to build our own parser. In fact, we need at least our own generated `JTBParser.java` file. Everything else can be taken from the standard distribution. For generating such a parser we use the `-tk` flag as shown above, which generates the special tokens into the syntax tree.

2.5 Design

Our Annotation Tool mainly has 3 tasks to do. On one hand, we have to read in the annotations provided by the Type Inferer. On the other hand, we have to parse 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 parsing of the annotations in XML file format is done by the XMLBeans [11] library. To be more precise, the XMLBeans library is used to generate the `xmltypes.jar` file based on the `annotations.xsd` XML Schema(see Appendix A). The xmltypes library is then used to parse the `annotations.xml` file containing the annotation information. For the java source file parsing, the Java Tree Builder in combination with the JavaCC has been chosen as described in section 2.4. As mentioned the Java Tree Builder provides us an interface implementing the visitor pattern. With a customized visitor we can explore and eventually modify the syntax tree. We can, of course, also output the syntax tree so that we get an exact copy of the input source file. This is the basis where the work of the Annotation Tool will be built on.

Basically, there seem to be two ways to combine the XML annotation reading and the java source parsing. One approach would be to go through the XML annotations and hand over the information to the syntax tree. In principle, this could be done either by inserting the information into the syntax tree or by adding the annotation information right while a visitor outputs the whole tree and so creates an annotated source file. Adding the annotations into the syntax tree is not that simple, because the tree is based on a pure java 1.5 grammar (see also 2.8). And since we are only interested in an annotated source file and do not need an annotated syntax tree, the insertion of the annotations into the syntax tree is not the way to chose.

But going through the XML annotation file and handing over the information right while outputting the syntax tree does not lead to the solution either. The information in the XML annotation file is unordered, which means it is not guaranteed that the annotations in the XML file are listed in the same ordering in which the visitor visits the corresponding elements of the syntax tree. This fact renders it impossible (at least without a significant pre-processing overhead) to insert all the annotations during one parse and output. Multiple parses and outputs in which only a part of the annotations get inserted would be an option, but is not possible at the moment either. The reason is that we only have a pure java 1.5 grammar, which does not recognize our modifier keywords and therefore will reject any java file already annotated with some Universe types (see also 2.8).

The alternative approach to processing the XML file annotation by annotation is explained in the following. It is basically the opposite of the first approach. After parsing the source file and creating the syntax tree, a visitor is run over the syntax tree and generates as output a source file. But in addition to just dump the unmodified source file, at tree nodes where annotations may be located the visitor requests the annotation information (if present) from the XML file. The advantage of this approach is that we do not have to modify the syntax tree itself. Furthermore, it is guaranteed that each java source file is annotated with only one parser invocation and one syntax tree dump of the proposed visitor. A negative point of this approach with focus to speed is, that at each position where theoretically an annotation can occur, we perform a request, even if there is no annotation to be inserted. But this overhead gets smaller the more complete the annotation information is.

2.6 Implementation

The Annotation Tool was developed in Eclipse and is written in Java. The JDK 1.5 was used. There is only one package called `ch.ethz.inf.sct.annotator` present in the Annotation Tool. It consists of five classes:

JTBParser.java (JTBParser and JTBToolkit class)

The JTBParser class is the main class of the JTB parser. All other classes belonging to the JTB are used directly from the distributed library (`jtb132.jar`). The standard JTBParser and JTBToolkit classes cannot be used because they discard special tokens, to which all the comments belong. Therefore a customized JTBParser and JTBToolkit had to be generated. For more information about the generation of these files see also section 2.4. For this project the JTBParser is constructed by passing a FileInputStream from a java source file, which is to annotate. The parser is started by invoking its static method `CompilationUnit()`. If we have more than one file to annotate the parser gets reinitialized by invoking its `ReInit(InputStream)`

method. As InputStream the FileInputStream of the next java source file is passed.

XMLAnnotationReader.java

The XMLAnnotationReader supports the AnnotationVisitor in reading the annotation information out of the XML annotation file. Due to the provided functionality of the XMLBeans library it is inconvenient for the AnnotationVisitor directly to read out the annotations. The XMLAnnotationReader as intermediate class does the traversing of the XML file and provides a simple interface to get the annotation information. Due to the fact that the XML annotation file is unordered and in order to keep the interface simple, we have to keep track of the current position of the AnnotationVisitor e.g. in which method the annotation process is at the moment. Therefore the XMLAnnotationReader contains some fields for the storage of references to the current class, method, field etc. These fields can be modified by invoking of the corresponding set methods.

Since the XMLBeans and the XMLTypes library respectively maps the XML annotation file into arrays of classes, methods, fields etc, reading an annotation means traversing an array. Currently, the corresponding array is traversed each time an annotation request is performed. If speed becomes an important factor, creating hashmaps the first time an array gets traversed could improve performance.

AnnotationVisitor.java

The AnnotationVisitor is a visitor for a syntax tree built up by the JTB parser. It extends the DepthFirstVisitor provided by the JTB library. Basically it acts like a tree dumper, which just produces the original java source file as output. But while traversing the syntax tree, the AnnotationVisitor not just dumps all the NodeTokens to the specified output stream, it also requests the annotations from the XMLAnnotationReader and inserts them into the output stream at the proper locations.

Due to the nature of some annotations, like for example annotations for type casts, we cannot identify them by a name or another lexicographic term. If we have several occurences of a type cast, we need a way to distinguish them. Therefore these types of annotations are referenced by an index instead of a name (see also Appendix A).

For the AnnotationVisitor this means, if it encounters a location where such a type of annotation could occur, it does not only have to request the annotation information. It also has to keep track of the number of occurrences, and should only annotate if the current number matches the index given in the XML annotation file. Therefore the AnnotationVisitor has some fields called xxxIndex (e.g. castIndex). Each time the visitor encounters a location where such

a type of annotation could occur, the corresponding index gets increased. So for example if the visitor meets a type cast the counter castIndex gets increased.

Most of the annotations are not very difficult to insert. Once the location and the corresponding annotation is found, it simply can be inserted without changing the structure of the source code significantly. However, there appear some difficulties if we have multiple field declarations on one line. Consider the following declaration:

```
T t1,t2;
```

If `t1` and `t2` get the same Universe Type modifier, it suffices to annotate their type `T` once. But in the case of different modifiers for `t1` and `t2`, it is not allowed to annotate just the type `T` since both,`t1` and `t2`, refer to this type:

```
/*ref*/ T t1,t2;
```

To surpass this, the program divides such multiple declarations into single declarations. So, the line is split up into

```
T t1; T t2;
```

These single declarations can then be annotated without problems. For example like:

```
/*ref*/ T t1; /*peer*/ T t2;
```

Annotator.java

This is the main class of the Annotation Tool. Here the different parts get connected. First it instantiates a JTBParser, a XMLAnnotationReader and an AnnotationVisitor. Second, it invokes the parser to build up the syntax tree, passes the XMLAnnotation to the AnnotationVisitor and starts that one in order to traverse the syntax tree. The second step happens for

each java source file that should get annotated. The usage of the Annotator class is as follows:

```
java ch.ethz.inf.sct.annotator.Annotator AF JF (JF)*
```

with:

AF The XML file with the annotation information. Normally, this file is obtained from a Type Inference tool (e.g. for the Runtime Type Inference tool this file is called annotations.xml per default). In principle, such a file can also be created manually. But please note that the Annotation Tool assumes proper input and does no structure or consistency checking. So, in order to work properly, this file must have the structure described in the XML Schema file annotations.xsd (see Appendix A)

JF This specifies one of the Java source files going to be annotated. The list of the source files does not have to be complete. It does not matter if not all of the involved Java source file are listed, even if there are annotations present for them in the XML annotation file. Neither it matters if we list source files which do not get any annotation at all.

2.6.1 Libraries

The libraries used for the Annotation Tool need to be contained in the classpath:

- jsr173_api.jar
- jtb132.jar
- xbean.jar
- xmltypes.jar

All these libraries are located in the `lib` directory of the project files. The xmltypes.jar file was created from the annotations.xsd XML schema using `scomp` from the XMLBeans library. The XMLTypes library will create classes according to the entry types of the XML schema. So, for each entry of an XML annotation file, an instance of the corresponding class gets created. The tree structure of the XML files is represented by arrays. That means, e.g. each class that represents a method has an array of classes that represent its parameters.

2.6.2 XML annotation file

The XML annotation file as obtained for example by the Runtime Type Inferer does not only contain the information for the annotations themselves. Moreover it contains additional information on how the annotations should look like or the kind of output produced. Let's have a look at the <head> element of the XML annotation file or consider the XML schema file annotations.xsd (see Appendix A):

ToolTarget

The `<target>` tag specifies what target should be modified. The options are java and jml. Choose java, if the annotations should be embedded into the original java source files. With jml the JML specification files get created.

ToolStyle

The `<style>` tag lets you define the style of the annotations itself. The options are as follows:

types - as standard type annotations, e.g. `"peer T"`

jml - as JML comments, e.g. `"/*@ peer @*/ T"`

oldjml - as escaped JML comments, e.g. `"/*@ \peer @*/ T"`

2.7 Results

In this section we want to present two examples of annotated programs. They show the different annotation cases. These examples (including the XML annotation file, the original files and the annotated files) are located in the project folder examples. The first one is the Linked List example from the Runtime Type Inferer. Listing 2.1 shows the annotated code:

Linked List

Listing 2.1: Annotated Code for Linked List example

```
public class LinkedList {
        /*@ rep @*/ ListItem head;
        int size;

        public LinkedList(){
```

```
                head = null;
                size = 0;
        }

        public int size(){
                return size;
        }

        public void callbackRequest(Object obj){
                if(head != null){
                        head.relayCallbackRequest(this, obj);
                }
        }

        public void callback(){
                System.err.println("Was called back");
        }

        public void insert(int i, /*@ peer @*/ Object o){
                if(head == null){
                        head = new /*@ rep @*/ ListItem(o);
                }else{
                        head.insert(o);
                }
                size++;
        }

        public boolean contains(/*@ peer @*/ Object o){
                if(head != null){
                        return head.contains(o);
                }else{
                        return false;
                }
        }

        public Object remove(/*@ peer @*/ Object o){
                if(head == null){
                        return /*@ peer @*/ null;
                }else if(o.equals(head.stored)){
                        Object ret = ((/*@ peer @*/ ListItem)head).stored;
                        head = head.next;
                        size --;
                        return /*@ peer @*/ ret;
                }else{
                        Object ret = head.remove(o);
                        if(ret != null){
                                size --;
                        }
                        return /*@ peer @*/ ret;
                }
        }
}

public class ListItem {
        /*@ readonly @*/ Object stored;
  /*@ peer @*/ ListItem next;
        public String name;

        ListItem(/*@ readonly @*/ Object toStore){
                stored = toStore;
                next = null;
        }

        public ListItem getNextItem(){
                return /*@ peer @*/ next;
```

```
        }

        public void insert(/*@ readonly @*/ Object toStore){
                if(next == null){
                        next = new ListItem(toStore);
                        next.name = "item";
                }else{
                        next.insert(toStore);
                }
        }

        public Object remove(/*@ readonly @*/ Object o){
                if(next == null){
                        return /*@ readonly @*/ null;
                }else if(o.equals(next.stored)){
                        Object ret = next.stored;
                        next = next.getNextItem();
                        return /*@ readonly @*/ ret;
                }else{
                        return /*@ readonly @*/ next.remove(o);
                }
        }

        public /*@ pure @*/ boolean contains(/*@ readonly @*/ Object o){
                if(o.equals(stored)){
                        return true;
                }else{
                        if(next == null){
                                return false;
                        }else{
                                return next.contains(o);
                        }
                }
        }

        public void relayCallbackRequest(LinkedList list, Object obj){
                if(stored.equals(obj)){
                        list.callback();
                }else{
                        if(next != null){
                                next.relayCallbackRequest(list, obj);
                        }
                }
        }
}
```

AnnotatorTest

The second example called AnnotatorTest is created mainly to show the annotation cases that are not present in the Linked List example (especially the static call annotations and the multiple line annotations). Listing 2.2 contains the original source file, while listing 2.3 shows the corresponding extract of the XML annotation file. The annotated source file is found in listing 2.4. Please note that the entries in the XML annotation file for this example were generated manually and not by an inference tool. This includes the concrete annotations to be chosen arbitrarily.

Listing 2.2: Source code for AnnotatorTest example

```
public class AnnotatorTest {
        Object o1 = new Object();
        Object o2,o3,o4;

        public static void main(String[] args) {

                SomeClass.println(        "SomeClass:static  call 0");
                AnotherClass.println("AnotherClass:static  call 0");
                SomeClass.println(        "SomeClass:static  call 1");
                AnotherClass.println("AnotherClass:static  call 1");
                SomeClass.println(        "SomeClass:static  call 2");
                AnotherClass.println("AnotherClass:static  call 2");
        }
}

public class SomeClass{
        static void println(String str){
                System.out.println(str);
        }
}

public class AnotherClass{
        static void println(String str){
                System.out.println(str);
        }
}
```

Listing 2.3: Extract of the XML annotation file

```
<class name="AnnotatorTest">
  <field modifier="rep" name="o1" type="java.lang.Object">
    < field_init >
      <new index="0" modifier="rep" type="java.lang.Object"/>
    </ field_init >
  </field>
  <field modifier="peer" name="o2" type="java.lang.Object"/>
  <field modifier="rep" name="o4" type="java.lang.Object"/>

  <method name="main" modifier="">
    < static_call index="0" modifier="rep" type="SomeClass"/>
    < static_call index="1" modifier="rep" type="SomeClass"/>
    < static_call index="1" modifier="peer" type="AnotherClass"/>
  </method>
</class>
```

Listing 2.4: Annotated source code for AnnotatorTest example

```
public class AnnotatorTest {
        /*@ rep @*/ Object o1 = new /*@ rep @*/ Object();
        /*@ peer @*/ Object o2; Object o3; /*@ rep @*/ Object o4;

        public static void main(String[] args) {

                /*@ rep @*/ SomeClass.println(        "SomeClass:static  call 0");
```

```
            AnotherClass. println ("AnotherClass: static   call 0 ");
            /*@ rep @*/ SomeClass.println(          "SomeClass:static   call 1 ");
            /*@ peer @*/ AnotherClass. println ("AnotherClass: static   call 1 ");
            SomeClass. println (          "SomeClass: static   call 2 ");
            AnotherClass. println ("AnotherClass: static   call 2 ");
        }

}
```

2.8 Future Work

2.8.1 Partially annotated programs

The Annotation Tool, as it is at the moment, can only handle pure java files. That means it works only on java source files that do not contain any annotations yet, since the used grammar is a normal Java 1.5 grammar. It would be useful if the Annotation Tool could handle partially annotated files. In order to solve this problem, the Java 1.5 grammar has to be extended so that it accepts the already existing annotations while parsing. In addition, one could imagine having some conflict solution response. The user would then be notified, if the existing and the new annotations are distinct.

Chapter 3

Runtime Visualizer

3.1 Motivation

With the Runtime Universe Type Inference project there exists a powerful tool to gain universe type annotations out of the runtime behaviour of a program. In order to make the process of computing these annotations more clearly, we would like to have it visualized. Therefore, it would be nice if we had a step-by-step graphical representation of the creation of the Extended Object Graph as it is done by the Runtime Type Inference tool. Since the success of finding appropriate annotation information depends strongly on the concrete runtime behaviour of the analyzed program, it would be desired to have a program execution as extensive as possible, meaning a program execution that covers as much of the code as possible. Often this is not the case, so the Runtime Type Inferer might not have enough information to decide on all the annotations. At that point, help from the user would be appreciated. Therefore, it would be helpful if the user could get a visualization of the Extended Object Graph and eventually interact with the Runtime Type Inferer in order to perform changes on the Extended Object Graph as it is built up.

3.2 Evaluation

Several technologies for visualizing graphs were considered:

- GraphViz[12] - graph visualization that takes descriptions of graphs in simple text language
- JGraph[13] - swing-compatible graph visualization library
- OpenJGraph[14] - library to create and manipulate graphs
- Graphical Editing Framework (GEF)[15] - a plugin for Eclipse

3.3 Decision

The development environment Eclipse contains a powerful graphical framework, which provides the needed functionality to interact with another program and continuously display a changing graph. Furthermore, an integration of the visualization into the IDE used by the Runtime Type Inference tool and the Annotation Tool could be a benefit. Therefore, Eclipse's Graphical Editing Framework was chosen as basis for the Runtime Visualizer.

3.4 Design

It is not intended by this report to give a complete guide for the Graphical Editing Framework. But for better understanding of the following sections, some basics of the GEF are introduced here. For more information about GEF see [15, 16].

The Graphical Editing Framework allows us to develop graphical representations for existing models. All graphical visualization is done via the Draw2D framework, which is a standard 2D drawing framework based on SWT from eclipse.org (see 3.1). For more information on SWT or Draw2D, see [17, 16, 15].

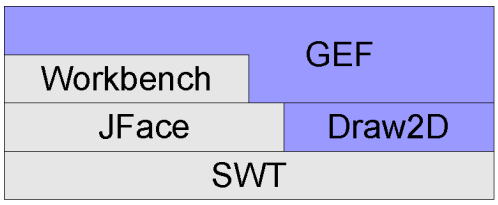

Figure 3.1: GEF dependencies

The editing functionality of the Graphical Editing Framework makes it possible to build graphical editors for almost every model. With these editors, simple modifications can be done to the model, like changing element properties or complex operations like changing the structure of the model. The linkage between a model and its graphical representation follows the Model-View-Controller paradigm. But contrary to the common scheme of MVC, in GEF there exists no direct link between the view and the model. A short explanation of Model, View and Controller (see also figure 3.2) and some other basic notions follow:

Model

An almost arbitrary model. In GEF, models should be changed using Commands. The model should know nothing or as little as possible about its view.

View

The view consists of the visual part, which is the primary representation for the model objects. For example, Figures from Draw2D are basic visual parts. The view also includes feedback, handles, tooltips, basically all the things visible to the user.

Controller

A Controller is called EditPart in GEF. Editparts connect the model with the view(s) and glue them together. EditParts are also responsible for modifications of the model. Normally, this is done with Commands.

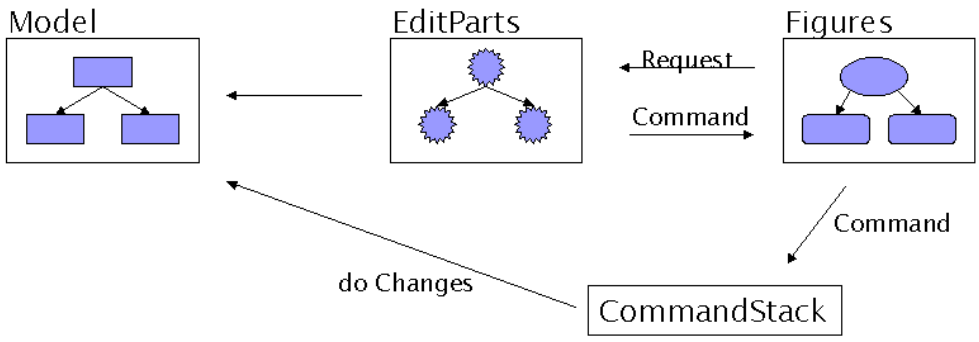

Figure 3.2: GEF structure

EditPolicy

An EditPolicy defines a special behaviour of an EditPart. This behaviour is basically defined by the Requests that the EditPolicy understands. Requests which do not apply to the EditPolicy are ignored. An EditPart handles Requests by iterating over all of its EditPolicies.

Request

Requests are the communication objects in GEF. They are used to communicate with an EditPart. If a Request is sent to an EditPart, the EditPart delegates this Request to its installed EditPolicies. These will as mentioned above, process the Request or ignore it if it does not apply to them.

Command

Commands are used to modify the model. Usually, Commands get created by the EditPolicy of an EditPart. After creation, Commands are sent to the CommandStack, where the Commands get executed.

CommandStack

All the Commands are sent to the CommandStack in order to get executed. The CommandStack provides also some undo/redo functionality.

3.4.1 The Runtime Visualizer as an Eclipse Plugin

As mentioned, the Runtime Visualizer is embedded in the development environment Eclipse. This is done by implementing the Runtime Visualizer as an Eclipse plugin. The visualization is presented in an Eclipse Editor. The visual part of the project also includes some action buttons and the Properties view of Eclipse itself, where information about objects in the graph will be available.

Figure 3.3 shows the major contributions from the Runtime Visualizer to the Eclipse Workbench. For more information on the Eclipse Workbench and Plugin development, we refer to the Eclipse help (also available online [16]).

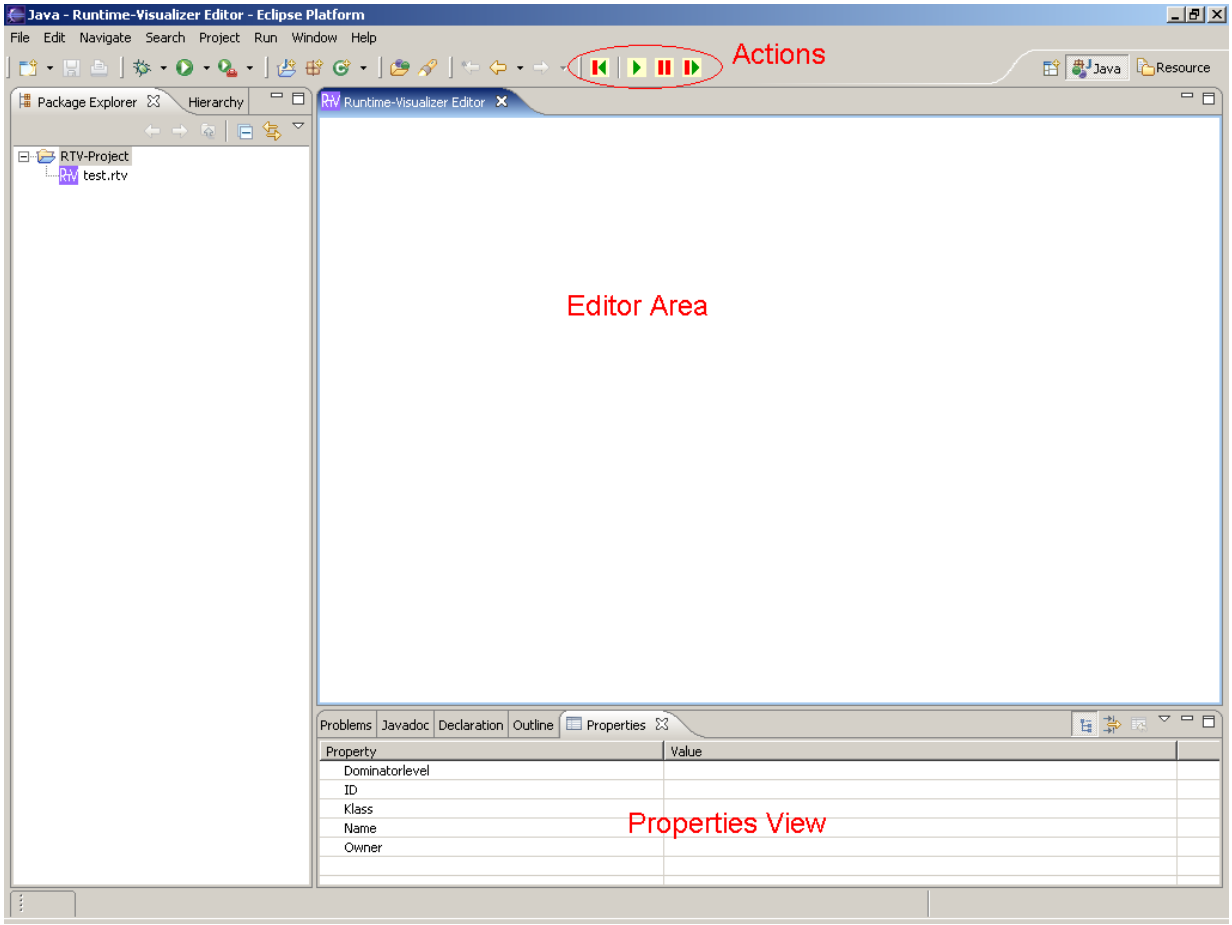

Figure 3.3: Eclipse workbench with Runtime Visualizer plugin

3.5 Implementation

For implementing the Runtime Visualizer project, Eclipse SDK 3.1.0 with Java SDK 1.5.0 and Graphical Editing Framework SDK 3.1.0 were used. The implementation follows the design of the GEF as closely as possible, but with some modifications due to our special model behaviour. The major difference is that our model does not only get modified by the user as in normal GEF applications, but in addition must get modified by the changes in the Extended Object Graph of the Runtime Type Inferer. In order to get notified of changes in the Extended Object Graph and to be able to do the corresponding modifications in the GEF model, the Runtime Type Inferer provides an observer interface. So, we can add a customized observer which gets notified of the changes in the Extended Object Graph and modifies the GEF model accordingly (see Figure 3.4). But since the thread running our observer notification methods is the thread of the Runtime Type Inferer, it is no UI thread. In fact this means that it cannot execute commands and directly perform modifications on the graph. Therefore the GraphCommandStack is a special implementation of a normal CommandStack where Commands can be queued up and executed by the UI thread of the GEF afterwards (see GraphCommandStack below).

While the process execution is in a notification method of an observer, the computations of the Runtime Type Inferer are suspended(like calling a blocking method). This enables us to halt the Runtime Type Inferer each time the observer gets notified. To control the process of the Runtime Type Inferer simple player functionality has been added in order to be able to pause and continue at each step of the generation of the Extended Object Graph. The visual representation of these player functions, called Actions, are shown in Figure 3.3 and for more information consider the actions package described below.

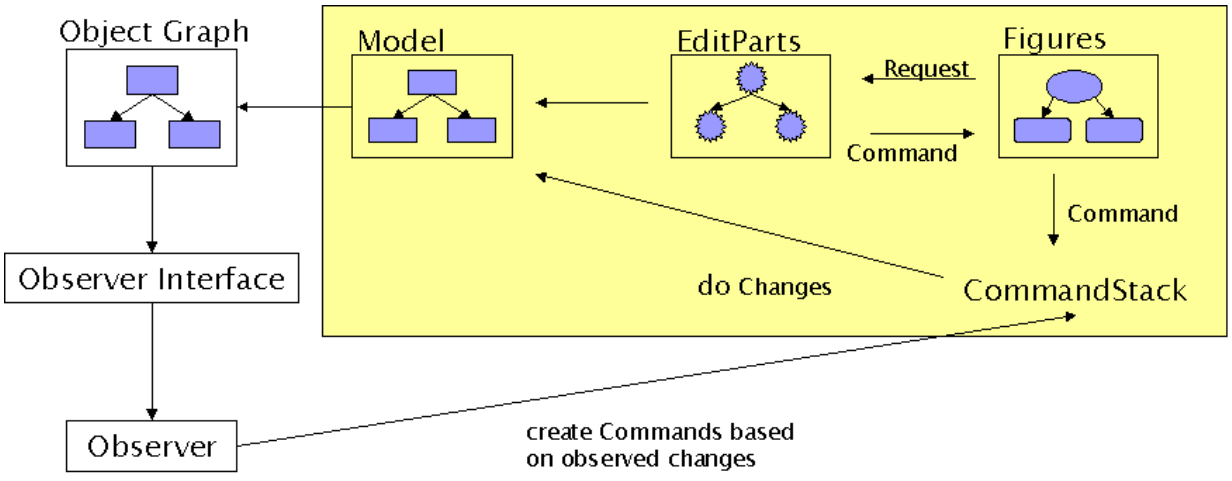

Figure 3.4: Runtime Visualizer

root package (ch.ethz.inf.sct.runtimeVisualizer)

The classes situated in the root package mainly are involved in the linkage between the GEF part and the Runtime Type Inferer part. Implementations of the observer, the command stack and the player functionality are located here. The RuntimeVisualizerPlugin class, the main class of the created plugin, can also be found in this package.

EOGObserver

The EOGObserver does the major work of changing the GEF model according to the Extended Object Graph. It is implemented as an observer of the Extended Object Graph and each time it gets notified, it induces the appropriate changes to the GEF model if necessary. Since the thread executing the notification methods comes from the Runtime Type Inferer and is no UI thread, it cannot change the GEF model directly. This complicates the model editing significantly. Therefore, the observer creates Commands which then are sent to the GraphCommandStack in order to be executed. The RefreshThread which is run at each notification then updates the view including the execution of all pending Commands in the GraphCommandStack.

GraphCommandStack

This class extends the normal CommandStack and provides an accessible queue where the Commands generated by the EOGObserver can be sent to. By refreshing the graph (like the RefreshThread does), this queue will get processed and the Commands stored there are executed. This customized command stack is needed due to the fact that only the UI thread can execute Commands. And since the thread which runs on the EOGObserver is no UI thread, direct command execution is not possible.

Player

The set methods of the Player class are called by the corresponding Actions. These Actions are triggered by the user in the UI (see package actions). The Player class maintains the player state and notifies the EOGObserver of state changes.

PlayerState

The class containing an Enumeration of the three player states PLAYING, PAUSED and ONESTEP.

RefreshThread

The RefreshThread is run by the EOGObserver in each notification method in order to update the view and to process the pending Commands. The RefreshThread invokes a refresh on the whole graph which includes invoking the `refreshGraph()` method of the GraphEditPart. There the pending Commands will get executed.

RuntimeVisualizerPlugin

This is the main class of the plugin for the Runtime Visualizer. It handles plugin activation, deactivation and a link to the UI icon of the plugin is provided here, too.

package actions

Here, two sorts of classes are located: The Action classes and the classes that contribute these actions to the UI. The Action classes are named like XxxAction where Xxx corresponds to the player function they activate. In such a XxxAction class, the look of the buttons and tooltips is defined. Moreover they have to implement the run() method which defines what is to be executed if the Action is triggered(e.g. by pressing the corresponding button on the action toolbar.

The RuntimeVisualizerActionBarContributor and RuntimeVisualizerContextMenuProvider are needed to make the Actions available to the user (see also Figure 3.3). While the task of the first one is to embed the actions into the ActionBar located in the toolbar of Eclipse, the second one inserts the actions into the ContextMenu. The ContextMenu is shown if you right-click with the mouse on the area of the editor.

package editors

The RuntimeVisualizerEditor is the only class present in this package, since we have only one type of editor for the Runtime Visualizer. This editor does most of the initializing of our GEF components. The GraphicalViewer gets configured, which includes setting the RootEditPart, EditPartFactory, ContextMenuProvider. The CommandStack and the EOGObserver are created here, too.

Furthermore, Eclipse editors contains by default some load and save functionality. To be precise, each editor instance corresponds to a file. This means, we can store at any time the currently displayed graph. This is done by serializing the current model. Since the model objects of the Runtime Visualizer contain references to the graph elements of the Extended Object Graph of the Runtime Type Inferer and these elements are not serializable, the Extended Object Graph will not be serialized. This means, that only the GEF model will be serialized. As a result, we can save and load at any time, but we cannot continue the type inferring process after a load because the model is not linked to the Extended Object Graph anymore.

package figures

In the figures package, classes for the visual representation of the model are located. Basically, this is the main part of the View (of the MVC pattern). An object of the Extended Object Graph is represented as a vertical expanding Container. It contains two horizontal expanding containers which represents its methods and fields. And of course, if the object is the owner of another object it contains this, too.

The package contains no special classes for representing the references between graph objects. Normal routed PolylineConnections are used which are managed by the corresponding EditParts of the connections.

package images

This package contains the image icons used for the buttons in the ActionBar and in the ContextMenu. These buttons correspond to the Actions of the player (e.g. PlayAction). See also the classes in the actions package.

package model

In this package, the classes representing the GEF model are to be found. Additionally, a Factory called ModelElementFactory is present, which is responsible for creating all the model elements. Basically, they represent almost a one to one mapping of the classes used in the Extended Object Graph of the Runtime Type Inferer. The difference is that all the classes of the GEF model are subtypes of the ModelElement class, so they form a tree structure.

package model.commands

The Commands used to modify the model are stored in this package. As explained, Commands normally get created by Policies. But in our case, since the GEF model has to match up with the Extended Object Graph of the Runtime Type Inference tool, most of the Commands get created by the EOGObserver. The observer sends them directly to the GraphCommandStack, where they get executed.

package parts

The parts package contains all the classes corresponding to the Controller part of the MVC pattern. These classes are named like XxxEditPart and reflect a one to one mapping of the model. So the ElementEditPart is the controller class corresponding to the ModelElement class of the model package. Additionally, a Factory called ContainerPartFactory is present, which is responsible for creating all the EditParts. The remaining classes were taken from the flow example[18] of the GEF distribution as they are. The BottomAnchor and TopAnchor classes provide the anchors where the arrows in the graph can be attached to. The GraphLayoutManager, DummyLayout and GraphAnimation classes are responsible of the layout of the graph combined with the animation in the graph. This motion can be seen between the states of the graph while playing (Player is set to play or onestep). The animation helps to understand what has been changed by the current step.

package policies

This package owns some basic Policy classes. They are needed to manage the highlighting of objects in the view or to allow objects to get deleted or moved (e.g. if the owner of an object has changed it has to be moved to the new context).

3.6 Results

The Runtime Visualizer shows the Extended Object Graph as it is built up by the Runtime Type Inferer. The user can pause and continue the build process with buttons emulating simple player functionality. Graphs can be saved at any step of the process and loaded for later consultation. The properties of an object in the graph can be accessed by left-clicking it with the mouse. The selected object's properties can then be seen in the properties view.

Figure 3.5 shows the workbench containing a completely built up graph of the Golf driver example from the Runtime Type Inferer. WriteReferences are shown in red while VarReferences are colored black.

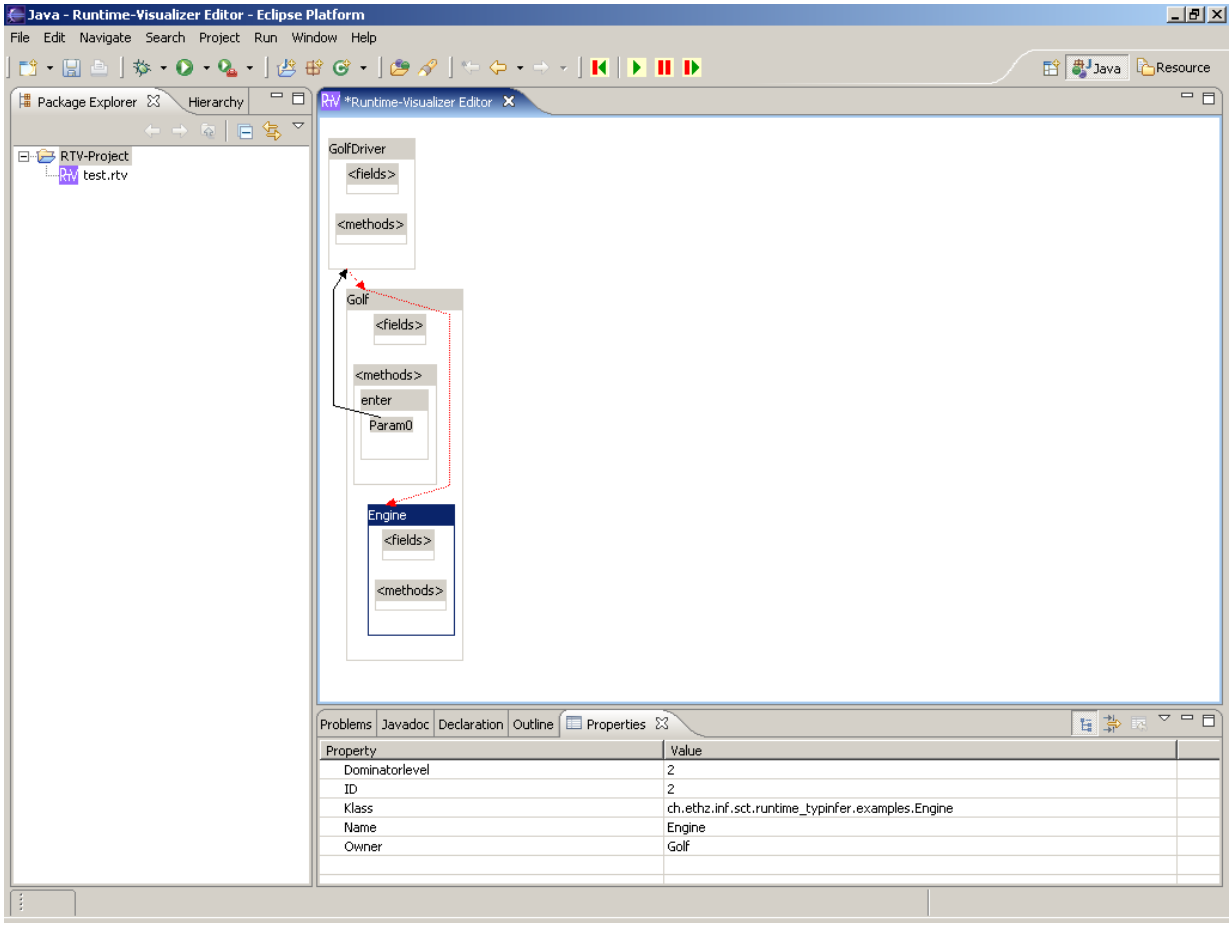

Figure 3.5: Snapshot of the Golf example

3.7 Future Work

3.7.1 Array Objects

As the Runtime Type Inferer evolves, the Runtime Visualizer should evolve, too. The work currently done on the Runtime Type Inferer by Marco Baer leads to the separate handling of arrays by the Runtime Type Inferer. Therefore, the Runtime Visualizer should be extended to visualize array types separately.

3.7.2 Visibility and Layout

Considering Figure 3.5 you might notice that even little programs can create space-consuming graphs. In order not to lose control over the graph, the visibility could be improved. Basically, there are two ways to achieve this. Improvements on the layout of the graph would increase visibility. Furthermore, the implementation of hide and show functionality would increase visibility and control drastically. Meaning one is able to collapse and expand single objects or the whole universe into smaller representations.

3.7.3 Interaction

The GEF framework was chosen in order to be able to alter the model. Next, some functionality to modify the model should be provided to the user. Such modifications could be done by drag-and-drop, palette functions or by additional buttons in the toolbar or context menu. GEF supports these possibilities. Note that the modification of the Extended Object Graph should be done with care, since we may easily lose consistency in the graph. Therefore, one might think of extending the interface between the Runtime Type Inferer and the Runtime Visualizer with appropriate modification functions which maintain consistency in the graph.

Bibliography

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

[2] W. Dietl and P. Müller. Exceptions in ownership type systems. In E. Poll, editor, *Formal Techniques for Java-like Programs*, pages 49–54, 2004.

[3] Nathalie Kelleberger. Static universe type inference. http://www.sct.inf.ethz.ch/projects/student_docs/Nathalie_Kellenberger/, 2005. Master Project.

[4] Frank Lyner. Runtime universe type inference. http://www.sct.inf.ethz.ch/projects/student_docs/Frank_Lyner, 2005. Master Thesis.

[5] Marco Baer. Practical runtime universe type inference. http://www.sct.inf.ethz.ch/projects/student_docs/Marco_Baer, 2006. Master Thesis.

[6] Antlr - another tool for language recognition. http://antlr.org.

[7] Jabstract: A full abstract syntax and parser for java. https://jabstract.dev.java.net.

[8] Cup - lalr parser generator for java. http://www2.cs.tum.edu/projects/cup.

[9] UCLA. Java tree builder. http://compilers.cs.ucla.edu/jtb.

[10] Javacc - java compiler compiler. https://javacc.dev.java.net.

[11] Xml beans. http://xmlbeans.apache.org.

[12] Graphviz - graph visualization software. http://www.graphviz.org.

[13] Jgraph - java graph visualization and layout. http://www.jgraph.com.

[14] Openjgraph - java graph and graph drawing project. http://openjgraph.sourceforge.net.

[15] Graphical editing framework 3.1. http://www.eclipse.org/gef.

[16] Eclipse 3.1 online help. http://help.eclipse.org/help31.

[17] Swt. http://www.eclipse.org/swt.

[18] Gef flow example. http://dev.eclipse.org/viewcvs/indextools.cgi/org.eclipse.gef.examples.flow.

Appendix A

Annotations XML Schema

```
<?xml version="1.0"?>

<!--
Schema for annotation files that specify Universe annotations that
should be added to existing sources.

Author: WMD

$Id: annotations.xsd,v 1.5 2005/05/12 12:35:46 dietlw Exp dietlw $
-->

<!--
TODO:
-
-->

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">


<!--XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Some additional types to automatically check the input.
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX-->

<!--
The modifiers that are valid for simple reference types.
-->
<xsd:simpleType name="SimpleUniverseModifier">
  <xsd:restriction base="xsd:string">
    <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="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>


<!--
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>


<!--XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
The elements of our schema.
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX-->

<!--
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 ref="head" minOccurs="1" maxOccurs="1"/>
```

```
      <xsd:element ref="class" minOccurs="1" maxOccurs="unbounded"/>
    </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="ToolTarget"/>

      <!-- What style of Universe annotations should we use? -->
      <xsd:element name="style" type="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 ref="field" minOccurs="0" maxOccurs="unbounded"/>

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

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

      <!-- Annotations for the class initializers. -->
      <xsd:element name="class_init" type="object_class_init"
   minOccurs="0" maxOccurs="unbounded"/>
    </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 ref="field_init" minOccurs="0" maxOccurs="1"/>
```

```
    </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 name="modifier" type="UniverseModifier" default="peer"/>
  </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 ref="return" minOccurs="0" maxOccurs="1"/>

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

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

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

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

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

    <!-- The name of the method. Multiple methods can have the
         same name, the parameters resolve the overloading. -->
    <xsd:attribute name="name" 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 name="modifier" type="UniverseMethodModifier" default=""/>
  </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 ref="new" minOccurs="0" maxOccurs="unbounded"/>

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

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

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

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

      <!-- The annotations for static calls in this method. -->
      <xsd:element ref="static_call" minOccurs="0" maxOccurs="unbounded"/>
    </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 name="modifier" type="UniverseModifier" default="peer"/>
  </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 name="modifier" type="UniverseModifier" default="peer"/>
  </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 name="modifier" type="UniverseModifier" default="peer"/>
  </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 name="modifier" type="UniverseModifier" default="peer"/>
  </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 name="modifier" type="UniverseModifier" default="peer"/>
  </xsd:complexType>
</xsd:element>


<!--
The annotation for a static method call.
The existing static method calls in a method are indexed, starting from zero.
-->
<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 name="modifier" type="SimpleUniverseModifier"
   default="peer"/>
  </xsd:complexType>
</xsd:element>


</xsd:schema>
```