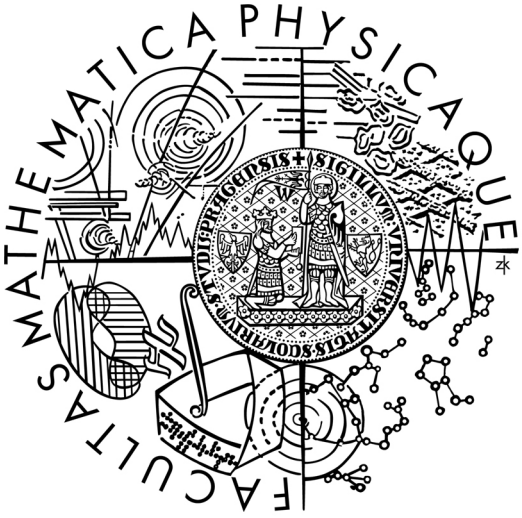
Faculty of Mathematics and Physics
Charles University, Prague

MASTER THESIS



Jiří Šebesta

XML Based Framework for Transformation of Java Source Code

Department of Software Engineering

Supervisor: RNDr. Tomáš Poch

Study Program: Computer Science, Software Systems

2009

I declare that I have worked out this thesis on my own, using only the resources stated. I agree that the thesis may be publicly available.

Prague, December 10, 2009 Jiří Šebesta

Content

Název práce: XML Based Framework for Transformation of Java Source Code
Autor: Jiří Šebesta
Katedra: Katedra softwarového inženýrství
Vedoucí bakalářské práce: RNDr. Tomáš Poch
e-mail vedoucího: Tomas.Poch@mff.cuni.cz
Abstrakt: Cílem této práce bylo vytvoření frameworku pro transformaci zdrojových kódů v jazyce Java. Základem frameworku je XML technologie. Framework definuje XML Schema použité k reprezentaci zdrojových kódů. Tyto zdrojové kódy (ve verzi Java 1.5) je možno za pomoci frameworku zparsovat a uložit do XML souboru. Prostředky pro transformace a analýzy kódu jsou založeny na technologii JAXB. Součástí frameworku je také několik ukázkových transformací (jako jsou refactoring a predicate abstraction). Z XML souborů je pak opětovně možné vytvořit zdrojový kód v jazyce Java.
Klíčová slova: Java, XML, transformace, framework

Title: XML Based Framework for Transformation of Java Source Code
Author: Jiří Šebesta
Department: Department of Software Engineering
Supervisor: RNDr. Tomáš Poch
Supervisor's e-mail address: Tomas.Poch@mff.cuni.cz
Abstract: Goal of the thesis was to provide transformation framework for Java sources based on well established XML technologies. The framework defines XML Schema for representation of Java sources. Java 1.5 sources can be parsed and stored to XML. Means for transformation and analysis are based on the JAXB technology. Several example transformations (refactoring, predicate abstraction) are also part of the framework. The framework is able to transform XML files back to the Java sources.
Keywords: Java, XML, transformation, framework

Chapter 1. Introduction

1.1. Motivation

Transformation frameworks enable transformations and analyses of source code. Transformations and analyses created by these tools can make the source code more (refactoring) or less (obfuscator) transparent. They can also help to find potential errors in the program and make the source code more effective. Transformation frameworks should have these features:

- Most of transformations need multiple iterations so there must exist an in-memory representation of the program.
- The representation should understand the programming language of sources to enable developing of effective transformations and analyzes.
- Transformation framework must be able to create in-memory model on the basis of given input and export the model back to the output format (after performing transformations).
- Framework can also contain means for cooperation with other tools during transformations.

There are several existing tools for transformations and analyses of programs (see the Chapter 2). All of them work directly with the source code or bytecode and use their own internal representation of program.

XML is well established technology and there are many useful tools that make working with the XML easier. Using these tools it's also possible to automatically generate classes for memory representation of the XML tree – these classes are generated on the basis of the schema of the XML file. It's not possible to create invalid XML document while using these classes.

Transformation framework based on the XML technology can use these advantages of the XML. Parsing the XML file (and building source tree in the memory) is faster than parsing of the Java sources. Java projects can be stored in the XML format until they must be compiled.

Chapter 2. Existing frameworks

Let’s summarize some existing tools for transformations and analyses of programs.

Recoder [4]

Recoder framework parses Java sources and creates abstract syntax tree (AST) – there is a Java class for each type of program element. While parsing the sources it analyzes references between program elements. After performing transformations Java sources are generated on the basis of transformed AST.

In-memory program metamodel is able to infer types of expressions, evaluate compile-time constants, resolve all kinds of references and maintain cross reference information. Recoder analyzes change impacts and automatically updates the model.

Recoder stores full information about the syntax of source codes including comments and formatting information so it’s possible to reconstruct the original source code.

It also contains small set of simple transformations and analyses. Probably the most useful transformation is Java 5 backporting. It enables to convert Java 5 (Java 6) source code to Java 4 compatible code.

Recoder is open source project but it’s not possible to compile sources using the standard javac compiler (Eclipse SDK is necessary to compile it).

JDT [7]

JDT is a large tool that is part of the Eclipse [9] platform. In fact, it is set of plugins – including user interface and debugging tools. Java program model and means for transformations and analyses are part of the JDT Core.

The Java program model is based on Java program element tree. It also contains indexed based search infrastructure that enables effective transformations (like refactoring that is also part of the JDT).

JDT parses Java sources to create program model and this model can be exported back to Java sources.

Core part of JDT can cooperate with other parts of JDT like graphic user interface or debugging tools.

Retouche [8]

Retouche is Java language infrastructure of the NetBeans platform IDE [10]. It wraps an instance of the JDK Java compiler and uses its abstract syntax tree. The classes that are used for representing program elements are part of the JDK 6. In addition, it enables quick access of the tree nodes and referencing within the AST.

Retouche works with Java sources – they are parsed to create the model and the model can be exported back to the Java sources.

This framework can also cooperate with the graphic user interface of the NetBeans platform.

Soot [11]

Soot is another type of tool – it doesn't work with source code. It's tool for transformations and analyses of Java bytecode.

It can be used either as a stand alone tool to optimize class files or as a framework. Developing transformations is not so easy because there is no abstract syntax tree and no information about the source code. However, Soot is an ideal tool for optimizing bytecode.

CIL [12]

CIL (C Intermediate Language) is a front-end for the C programming language. It parses a C program and compiles it into simplified subset of the C programming language so it's much easier to perform transformations and analyses over CIL program than over the original C program.

Many modules that use CIL for program analyses and transformations are also part of this project (e. g. liveness analyses, inliner, control-flow graphs).

GIMPLE [13]

GIMPLE is a simplified subset of GENERIC that is an intermediate representation language used by the GNU Compiler collection (GCC) as the "middle-end" when compiling source code. This form is common for all programming languages supported by the GCC.

GIMPLE is a three-address representation – all expression must have at most three operands. The GIMPLE tree is analyzed and optimized by the GCC to create more effective binary code.

Summary

These tools have many advantages and disadvantages. The XMLJT should combine as many advantages as possible.

It should have understandable in-memory AST structure like Recoder or JDT. Means for developing effective transformations and analyses (quick searching within the AST, references between tree nodes) must also be part of the framework (all of recently mentioned tools have such information).

Only JDT and Retouche have graphic user interface. However, both of them are very closely connected with it. The XMLJT should have graphic user interface but it also must be independent on the GUI (GUI should be built over the framework, not reversely).

Soot and CIL tools contain also many program analyses (e .g. control-flow graphs, point-to analyze) – the XMLJT should be also able to perform these analyses. It can use external tools to manage it. The XMLJT tool should also contain sample transformations like many recently mentioned tools do (JDT, Recoder, Retouche etc.).

Chapter 3. XML Technology

XML (eXtensible Markup Language) defines set of rules for encoding semi-structured documents. The language is defined by the W3C organization in the Specification of XML 1.0. XML is a textual data format using the Unicode encoding. Many languages have been developed on the basis of the XML (XHTML, RSS etc.).

XML documents have tree structure where the nodes of the tree are called elements. Each element can have any number of attributes and child elements. See the XML specification to get more information about the syntax and semantic of the language.

There are many tools and technologies connected with the XML. Let's summarize some tools and technologies that can be used by XML-based transformation framework:

XML schema

XML-based transformation framework can use one of XML schema definition languages to define valid XML representation of Java source code.

There are two common languages that define set of rules that the XML document must satisfy to be considered as valid. DTD (Document Type Definition) is the older one. It enables to define structure of XML documents but it's not possible to define any additional constraints (like number of child elements).

XML Schema (sometimes also called XSD - XML Schema Definition) is much more powerful than DTD. User types and many types of constraints can be defined using XML Schema. It also supports references between XML elements. XML Schema uses XML to describe the schema and it is now the most popular XML schema definition language.

In-memory representation

Transformation framework needs a tool to create in-memory representation of the XML tree.

The Document Object Model (DOM) is a platform and language independent object-oriented representation of a XML (or (X)HTML) document. The DOM enables to access the XML document using the corresponding object tree.

Java Architecture for XML Binding (JAXB) enables mapping of Java classes to XML document. Java objects can be created by unmarshalling (deserialization) of the XML representation. XML representation can be conversely generated by marshalling

these objects. Java classes can be automatically generated on the basis of XML Schema by the *xjc* tool.

Transformations

There are also existing tools that enable transformations of XML documents.

XSL Transformations (XSLT) is a declarative language based on the XML. It enables to transform XML documents. It doesn't change the original document but creates a new transformed document. This document doesn't have to be a XML document (can have any format like HTML or plain text). The XSLT processor is necessary to perform the transformations.

Chapter 4. Goal of the thesis

Goal of the thesis is to implement framework for transformations and analyses of the Java 1.5 source code. This framework is named XMLJT (XML Java Transformations). The framework will be based on well established XML (Extensible Markup Language) technologies. Several example transformations will be part of the thesis to proof the usability of the framework. The thesis will also contain instructions and tutorials for the user who would like to implement any transformation or analysis using the XMLJT framework. The graphic user interface extension of the XMLJT framework will make performing and testing of the transformations more comfortable.

The other part of the goal is to inspect capabilities of the XML technology for developing transformation frameworks. Advantages and disadvantages of this approach should be summarized.

Chapter 5. XMLJT Framework

5.1. Structure

The framework works with three representations of Java programs. Two of them are stored in file(s) (Java source code and XML) and one is stored in the memory (JAXB classes). The structure of both, XML and in-memory representation is precisely defined by XMLSchema. For details see the Chapter 5.2. The Figure 1 shows the structure of the XMLJT.

In-memory representation

It was possible to create own structure for representing the Java program in the memory. However, it would be quite difficult. The JAXB tool is able to generate Java classes on the basis of the XML Schema. Using this tool it's easy to create the in-memory source tree from the XML file (reverse transformation is also ensured by the JAXB tool). It's not possible to generate invalid XML document while using the JAXB classes. For details see the Chapter 5.5.1.

Means for transformation and analysis

Means for transformation and analysis are built over the JAXB classes. There are also several global objects that hold information about the JAXB tree to enable developing effective transformations. For further information see the Chapter 5.5.

Input and output

The framework doesn't work directly with the Java sources but it must be able to transform them to the XML format and back to the Java. The Java sources are parsed using the Recoder [4] tool. For further information about parsing see the Chapter 5.3. After performing transformations and analyses the XMLJT can export the memory representation of the program back to the Java sources (see the Chapter 5.4). It would be also possible to create XSLT transformation that would generate the Java sources directly from the XML file but this transformation doesn't exist yet. In fact, there are two parallel transformation frameworks. The first one is based on XSLT and the other one on the in-memory representation. We will focus on the in-memory based transformation framework that is more powerful.

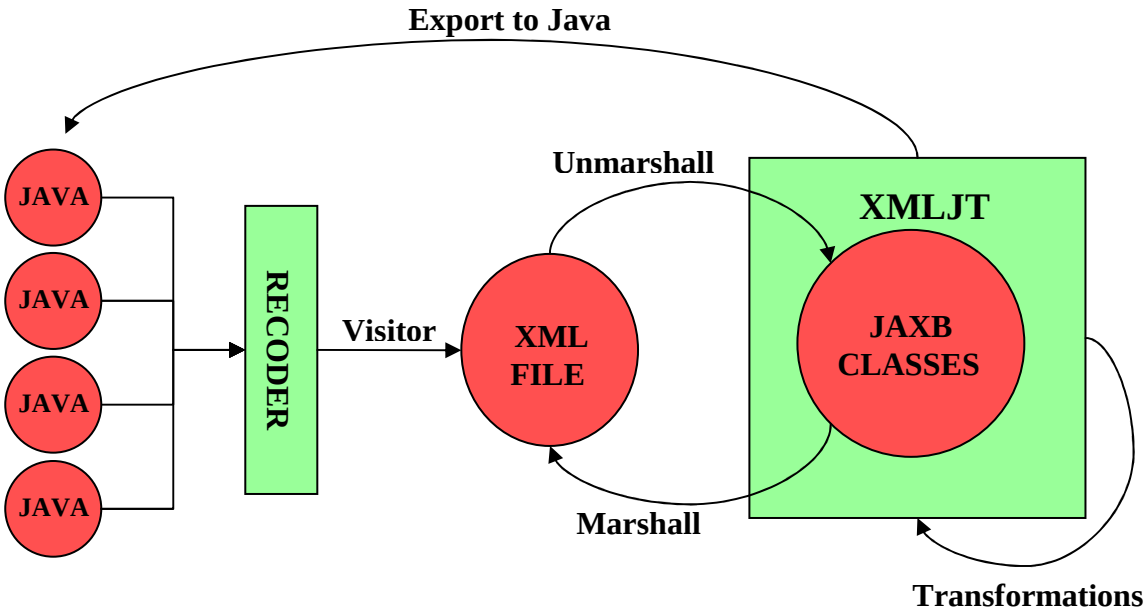


Figure 1: Structure of XMLJT framework

5.2. XML-based model of Java sources

5.2.1. JavaML

It was necessary to define format of XML files representing the Java sources. There are many possibilities how to manage it. DTD (Document Type Definition) and XML Schema are the most common ways to define schema of XML documents.

XML Schema has got many features that can make the work easier. Referencing (that enables defining relations between nodes) is its biggest advantage over DTD. XML Schema can also cover much more Java semantic rules than DTD can. Considering these facts XML Schema is used to describe the structure of XML documents. In-memory representation classes can be also generated on the basis of XML Schema (see the Chapter 5.5.1).

The schema didn't have to be created from the scratch. There was already one existing as the part of JavaML [1] project. This schema must have been modified.

5.2.2. Modifications of JavaML XML Schema

Schema of JavaML covers most of basic characteristics of the Java programming language. However, new elements had to be added to achieve full support of Java 1.5 and make the framework aware of classpath declarations:

Support of Java 1.5

JavaML schema doesn't support some Java 1.5 features (even *assert* keyword added in Java 1.4) so some new elements had to be added to the schema:

Annotations:

declaration – element *annotation* – contains list of properties' declarations

declaration of property – element *annotation-property-decl* (containing modifiers, name of property and its default value)

annotations usage – element *annotation-use* – all declarations that can be annotated in Java can contain this element

JAVA:

```
@interface MyAnnotation {
   int myProperty = 0;
}
……
@MyAnnotation
interface AnnotatedInterface {
……
}
```

XML:

```
<annotation id="93" idkind="type" name="MyAnnotation">
  <annotation-property-decl id="99" idkind="property" name="myProperty">
    <type dimensions="0" id="-65" idref="-14" name="int"/>
    <literal-number id="-66" kind="integer" value="0"/>
  </annotation-property-decl>
</annotation>
……
<interface id="91" idkind="type" name="AnnotatedInterface">
  <annotation-use id="-58" idref="93" name="MyAnnotation"/>
  ……
</interface>
```

Enumerations:

declaration – element *enum* containing declarations of enum constants and other declarations allowed by the Java 1.5

declaration of constants – element *enum-const-decl* contains name of a constant and parameters of its constructor (optional)

using constants in expressions – enum constants can be used as an expression anywhere in the source

JAVA:
```
enum MyEnum {
  MY_CONST;
}
……
MyEnum e = MyEnum.MY_CONST;
```

XML:
```
<enum id="102" idkind="type" name="MyEnum">
  <enum-const-decl id="107" idref="-69" name="MY_CONST"/>
</enum>
……
<var-initializer id="-102">
  <field-ref id="-103" idkind="field" idref="111" name="MY_CONST"/>
</var-initializer>
……
```

Enhanced for:

element *loop* can have attribute *kind* declared as “enhanced for”

Assert:

element *assert* containing an *expression* element

JAVA:
```
assert false;
```

XML:
```
<assert id="-54">
  <literal-boolean id="-72" value="false"/>
</assert>
```

References

Transformation needs to know references within the source code tree. For example it has to be able to find declaration of a local variable where a new value should be assigned. There can be many local variables having the same name and it’s not simple to find the right declaration. Once this declaration is found the reference is stored to XML and it can be reused in the future.

XML Schema supports referencing within the XML using *id* and *idref* attributes of XML elements. Types of all ids were changed to xs:int instead of xs:string so the searching can be more effective. All elements in the new schema must have id to enable direct accessing to the nodes of the in-memory tree. In the JavaML schema only declarations must have id.

References to classpath declarations

There can be several references to a type (method/field) that is not declared in the current XML (types declared in classpath). However, elements referring to them must have a valid value of the *idref* attribute (for example transformation needs to know that declared method overrides it's implementation contained in the classpath). There were two possible ways how to solve it. All references to outer types (methods/fields) may have the same *idref* (defined by constant) so it can be easily decided if the reference points to an existing element. But it would not be possible to decide if two references point at the same outer type. The first one can refer to a type, the other one to a method and they'll have both the same value of the *idref* attribute. To avoid this problem special tag named *outer-declaration* was defined. It has only two attributes: id and name (full name of type/method/field; name of method includes its signature). Negative ids are used for these tags so the user can easily see whether reference points to one of these outer declarations or to a real declaration within the document. Now the user can be sure all references with the same value of the *idref* attribute point at the same element.

```
JAVA:
String s = "abc";

XML:
<outer-declaration id="-22" name="java.lang.String"/>
……
<type id="-21" idref="-22" name="String" primitive="true"/>
```

Renaming *send* element

Element *send* of JavaML schema has been renamed to *method-call*. This element represents invocation of a method so the new name is more predicative than the original one. This modification wasn't necessary but it has made the schema more understandable.

5.3. Model creation

There are two ways to create the program model. It can be created either programmatically (using factory methods – see the Chapter 5.5.2) or by parsing Java sources. The next two subchapters describe how the model is being created from Java sources.

5.3.1. Parsers

It's necessary to parse the Java source code to be able to transform it to the XML format. There are plenty of available parsers so it has no sense to implement the new one. Let's summarize some parsers that might have been used and clarify our choice.

ANTLR v3 [5]

ANTLR (ANother Tool for Language Recognition) is a language independent parser and offers grammars for many programming languages including Java. Although the Java sources can be parsed easily, the generated parser does not provide resolving of references. The resolution must be implemented manually with respect to Java semantics and visibility rules. It's quite comfortable to work with this parser but its universality is not an advantage for us. If we wanted just simple Java to XML conversion, it would be no problem. But we would like to support transformations so we will need some additional information that is specific for the Java (for example for refactoring needs). We can find this information out ourselves but it would be superfluous work.

Javacc [6]

Javacc (Java Compiler Compiler) is the most popular parser generator for the Java. The Javacc also supports tree building via JJTree tool that is part of the Javacc. It would be probably the easiest way to use this parser but we decided to use more power tool that would afford more information about the source code (especially resolving of references).

JDT [7] and Retouche [8]

JDT and Retouche are powerful (supporting also resolving of references) tools working with Java source codes in well known platforms – JDT is part of the Eclipse platform [9] and Retouche is provided by the NetBeans platform [10]. Both tools are closely connected to their parent platforms and offer many capabilities to cooperate with them (e.g. cooperating with GUI) – dependence on these platforms is not an advantage for us.

Recoder [4]

Recoder is a framework having quite large functionality. It's not only the parser but it also delivers the whole infrastructure for Java sources analysis and transformation:

- Parsing and unparsing of Java sources
- Name and type analysis for Java programs
- Transformation of Java sources

Even we will use only the first of these features we decided to use this framework. The main reason was quite detailed information about the Java source code we get from this parser (will be described later in detail).

Our choice

Recoder tool was chosen to be used to parse the Java sources. The main reason was the support of references' resolving. Recoder is an alternative tool to the XMLJT (see the Chapter 2) but we use only its parser and it does not influence the rest of the framework.

5.3.2. Integration of Recoder into XMLJT

To convert Java to XML two parameters have to be defined: Java file(s) to be parsed and output XML file. There is also one optional parameter – classpath containing external libraries used by the sources to be parsed. This classpath is passed to the Recoder – wrong classpath can cause parse error. All input Java files are passed to the Recoder to be parsed.

Once the source code has been parsed the visitor design pattern (class *xmljt.java2xml.XMLTransformer*) is used to go through the Recoder's source tree and to generate XML file. DOM is used to create XML file. All references between XML elements (defined in XML Schema) are generated on the basis of references within the Recoder's source tree. The visitor implementation also generates ids for all XML elements. Declaration elements have positive id, all other have negative id.

5.4. Model export to Java

Generating Java source files is necessary for compiling the transformed program. This simple transformation just goes through the source tree using the visitor pattern and generates Java files. See the User manual (Appendix B) for instructions how to use it.

5.5. Model in-memory representation

The source code parse tree is the base of the memory representation of the Java program. The tree corresponds to the elements' tree of the XML file. It would be possible to perform transformations and analyses using just the information from the tree but it would be uncomfortable and ineffective. For example if the user knows the *MethodCall* node (representing an invocation of a method) and needs to find the method that will be called he must go through the whole tree to find the right node.

The *TreeInfo* class keeps information about the structure of the source tree and makes working with the tree much more effective (see the Chapter 5.5.2) – there is a reference table that maps ids of elements to references to the correspondent tree nodes. It also contains reference to an instance of the *ProgramElementFactory* class that can create new program elements.

There are some useful methods independent on the current *TreeInfo* instance so they can be declared as static. These methods can be found in the *TreeUtils* class and you can find their detailed description in the Javadoc. The Figure 2 shows structure of the in-memory program model representation.

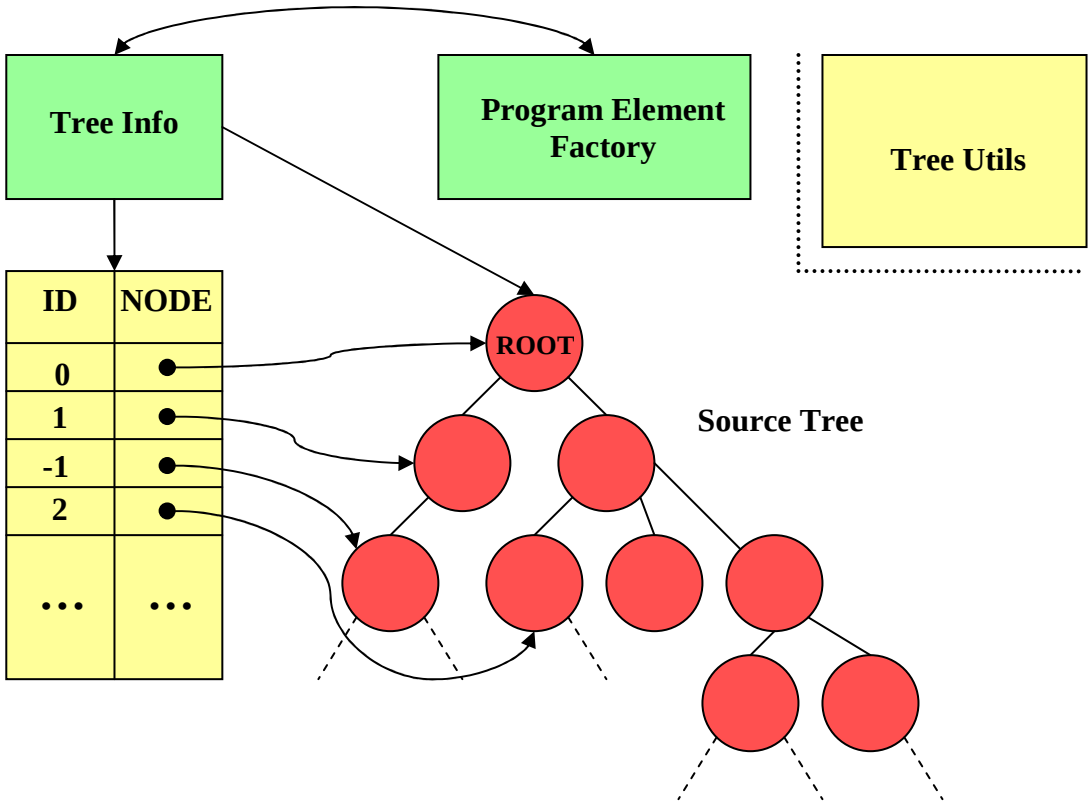


Figure 2: In-memory source code representation

5.5.1. Tree nodes representation

This chapter describes an implementation of the tree nodes classes' functionality. For information about using these classes see the Javadoc of the classes contained in the *xmljt.xmltree.nodes* package.

JAXB binding

In-memory representation of the source tree should correspond to the structure of the XML Schema. There must be a class for each type that is defined in the XML Schema. The JAXB technology is used for marshalling and unmarshalling of XML files. The JAXB needs the classes to have specifically annotated their fields and methods.

It is possible to write the JAXB classes manually but it's much better to let them be generated automatically. The JAXB creates these classes following given XML Schema. Unfortunately generated code is not as user-friendly as it should be. For example all child elements are mostly stored in one list – it's possible to get the list of all children of the class declaration element but there is no getter for the list of the methods' declarations, fields' declarations etc.

There are two ways how to solve it:

- Generated classes can be used as they are. In this case it would be necessary to create many utilities outside the generated classes. Taking functionality out of the classes would violate principles of the object oriented programming. Not modifying generated classes is the only advantage of this option.
- Generated classes can be extended so every change in the XML schema must be reflected in the source code of the framework. This disadvantage isn't big complication because the schema of Java programming language won't change frequently.

The second option has been chosen – the generated classes are extended. These extensions are placed in the *xmljt.xmltree.nodes* package.

Renaming generated classes

By default JAXB constructs classes' names from the names of XML Schema elements. For example *ClassDeclaration* class is generated from the <*class-declaration*>

element. However these names are straightaway, it would be better to name their extensions using this template. So the JAXB must generate classes with different names.

The JAXB offers two possibilities how to set binging options. The user can either add special tags to the XML Schema or pass the settings in a separate binding file. It's better to use external binding file because we don't want to change XML Schema any more. In this file (which is having XML format) JAXB is told to add "Element" suffix to all generated classes that correspond to some XML Schema element. It means *ClassDeclarationElement* class is now generated from the <*class-declaration*> element and its extension can be named *ClassDeclaration*.

Parent references in the JAXB tree

The tree of generated classes must be always searched from the top to the bottom because the nodes have no references to their parents. This feature can make many transformations ineffective so it must be changed. We can quite easily generate field (including getter and setter) and method that is called by the JAXB after marshalling and sets the actual reference to the parent (see the "JAXB plug-in" subchapter). However, there is no way to automatically update these references after any change within the tree. Reference to the parent must be actualized by methods that modify the tree. All setters, adders and removers in the framework classes do this automatically.

JAXB plug-in

References to the parent have to be added to all source tree classes. The source code can be added manually to the extended classes. Considering this code is identical in all classes it would be better to let the JAXB generate it. It's necessary to write JAXB plug-in to manage it.

Using this plug-in *parent* field is added to all classes that correspond to any XML Schema element. Getter and setter are also generated for this field. The last step is to generate *afterUnmarshal* method that is called by the JAXB after unmarshalling has been finished. The JAXB passes reference to the parent node as an argument of this method.

5.5.2. Tree info

The *TreeInfo* class keeps information about the structure of the source tree and makes the transformations more effective. For further information about using concrete methods see the Javadoc.

Direct access to the tree nodes is provided by the reference table. It contains two maps. The first one maps id to the node having this id. The other one maps id to the list of nodes that refers to the node with this id (for example id of method's declaration points to the list of all invocations of the method).

The *TreeInfo* class contains list of used ids and methods generating new unique ids (either positive for declarations or negative for other nodes). That's the reason why the element factory (class *ProgramElementFactory*) can't be static – it must use the *TreeInfo* instance to get new ids. *ProgramElementFactory* is created automatically while creating new instance of the *TreeInfo*.

5.6. Model transformation

This subchapter describes basic means for transformations and analyses that are part of the XMLJT framework.

The user needs to analyze (transform) the AST so the visitor design pattern will be the most common approach. There are two visitor classes. The first one is named *SourceVisitor* and can be found in the *xmljt.xmltree.nodes* package. It contains visitor methods for all types of tree nodes. These methods have empty bodies. The other one is *VisitAllElementsVisitor* (in the *xmljt.xmltree.treeinfo* package) – it extends the *SourceVisitor* and enables to visit the whole tree – default implementation of each visitor method visits all children of visited element. This visitor can be extended to go through the whole tree and transform some nodes (whose visitor method will be overridden).

The user also usually needs to create new AST elements. The *xmljt.xmltree.treeinfo.ProgramElementFactory* class ensures it. It contains factory methods for creating all types of AST nodes.

5.7. Graphic user interface

The XMLJT GUI makes implementing, testing and using of some transformations more comfortable. The user can see the Java source code and perform transformations.

It’s an alternative to the command line use of the XMLJT. While running transformation from the command line the user needs to specify XML elements by their ids. In the GUI he can easily select element(s) of source code (by selecting text in editor) and run the transformation (some additional parameters of transformation can be set in displayed dialogs).

It’s not possible to edit source code manually (only transformations can change it) so it cannot be used for developing Java programs.

5.7.1. Implementation

The graphic user interface is based on the Swing technology [14]. Jsyntaxpane [15] library is used for displaying the source code. This extension of standard editor pane supports syntax highlighting, searching within the file, displaying line numbers and some more useful features.

There must be a mapping between tree structure and plain source text. This map is being created while exporting program model to Java sources. It maps id of element to the corresponding segment of the Java source code. It enables to select tree elements by selecting text within the source code editor.

The extensibility of the GUI is ensured by the common interface for all GUI transformations. The extensions are loaded from the jar files using the Java reflection (name of extensions’ classes are stored in a text file).

Chapter 6. Proof of the concept – transformations

Several transformations based on the XMLJT transformation framework have been implemented to proof the usability of the framework.

Refactoring (the Chapter 6.1) is set of transformations – it contains some simple transformations like renaming and also few complex refactoring operations like inlining or method extracting. Predicate abstraction transformation (the Chapter 6.2) demonstrates possibility of cooperation with external tools (theorem prover).

6.1. Refactoring

The XMLJT framework contains some basic refactoring transformations. They can be used either separately or as the part of more complex transformations. See the User manual (Appendix B.6) for information about using these transformations.

6.1.1. Method extracting

This refactoring feature replaces block of statements by calling a separate method. Let's have a simple example:

```
1   int a = 3, b = 4;
2   a = b*a;
3   int c = a + b;
4   System.out.println(“result is ” + c);
```

Extracting method from the statements 2 and 3 will produce this result:

```
int a = 3, b = 4;
int c = extractedMethod(a, b);
System.out.println(“result is ” + c);

public int extractedMethod(int a, int b) {
  a = b*a;
  int c = a + b;
  return c;
}
```

The statements have been replaced by calling the newly created method. The result of the program must not change. Method can't be extracted from any sequence of statements because Java methods can return at most one value. Of course it would be

possible to return array or structure but it would complicate source code and be counterproductive to the refactoring.

Implementation

The user must enter statements that should be transformed and the name of method that should be created. At first it has to be verified that all selected statements follow each other within one block. Let's call this sequence of statements *block*. All variables that are read inside the block and have been initialized before the block must be passed to the extracted method (using its parameters). At most one variable that is set within the block can be read after the block (method can return only one value).

The result of calling method (if there is any) can be used to initialize local variable (like in the example) or be assigned to an already existing variable.

Body of the extracted method can't be the exact copy of the block. It is necessary to add declarations of variables that have been declared before the block, are never read within the block and are at least once set within the block (they don't have to be passed through the parameter but they have to be used in the method's body). Return statement (if there is some return value) must also be added. The last step is to change all references from the original variables to the newly created parameters.

If the block is part of a static method, extracted method must be also static.

6.1.2. Renaming

Renaming is a simple refactoring transformation that changes name of a program element (including all connected elements – for example renaming of a class will cause renaming of its constructors). Renaming of these declarations has been implemented:

- class
- interface
- enum
- enum constant
- field
- method
- parameter
- type parameter
- local variable

- annotation
- annotation property

Implementation

The only parameter is a new name of the declaration. Implementation hasn't been so difficult because of the support within the xml tree – program elements refer to the connected elements (for example the variable access points to its declaration). Sometimes the name of the declaration cannot be changed (outer declarations can't be renamed – access to the source code of the declaration is necessary).

Renaming of methods is a little bit more complicated. All connected methods have to be renamed too – by "connected" we mean:

- sub methods
- super methods
- all methods that have same ascendant as renamed method

6.1.3. Change method parameters

The user can remove method's parameter, add new one or change order of existing parameters. If new parameter is being added, its default value must be defined - this value will be used as an argument everywhere the method is being called. Permutation of the parameters' order will cause the same permutation of arguments everywhere the method is being called.

Implementation

It is necessary to change the parameters of the method declaration and change the appropriate arguments everywhere the method is being called. The same transformation must be performed on all connected methods (sub methods, super methods… - see the renaming implementation).

6.1.4. Variable to field

This simple transformation just replaces local variable by a newly created field so it can be accessed from other methods of the class.

Implementation

New field is created, local variable declaration is removed and all references to this variable are replaced by the references to the newly created field. If the variable is part of a static method, the field is also declared as static.

6.1.5. Extract interface

Extracting an interface A from a class B means creating new interface that contains declarations of some (or all) methods that are implemented in the class B. If the user has got:

```
public class B {
  public void methodA() {
     System.out.print("A");
  }

  public int methodB(String s) {
     System.out.print(s);
     return 0;
  }
}
```

and wants to extract an interface containing the declaration of the *methodB*, he will get:

```
public class B  implements A {
  public void methodA() {
     System.out.print("A");
  }

  public int methodB(String s) {
     System.out.print(s);
     return 0;
  }
}
```

```
public interface A {
  public int methodB(String s);
}
```

Implementation

The user must enter three parameters: name of an interface that should be created (including package), class that is the source of the extracting and all methods that should be declared in a new interface. All methods must be checked to have correct modifiers – Java allows only public (or having the default access modifier) non-static methods to be declared in an interface.

After creating the method headers in the interface all necessary imports have to be added to the newly created file (using method *getNeededImports* of the *TreeInfo* class – see the Javadoc). The last step is to change the declaration of the source class to implement the new interface.

6.1.6. Extract super class

If the user wants to extract whole methods (not only their headers) from a class, he will use this refactoring transformation. Some methods can be extracted including their bodies and some methods can be extracted as abstract (their bodies will stay at the original positions). Let's have:

```
private class B {
  public void methodA() {
     System.out.print("A");
  }

  public int methodB(String s) {
     System.out.print(s);
     return 0;
  }

  public void methodC() {
     System.out.print("C");
  }
}
```

The user would like to extract the *methodA* (as abstract) and the *methodB* to a super class A – he will get:

```
public class B extends A {
  public void methodA() {
     System.out.print("A");
  }

  public void methodC() {
     System.out.print("C");
  }
}
```

```
public abstract class A {
  public abstract void methodA();

  public int methodB(String s) {
     System.out.print(s);
     return 0;
  }
}
```

Implementation

The user must enter four parameters: name of a class that should be created (including package), the class that is the source of extracting and all methods that should be declared in the new class. These methods are contained in two lists – the first one is the list of methods that should be moved including their body and the other one is the list of methods that should be declared as abstract in the super class (their body will stay at the original position).

The implementation is quite similar to the implementation of an interface extracting (see the previous subchapter). Static method can be moved to super class but cannot be declared as abstract. Private methods cannot be moved to the super class at all.

6.1.7. Anonymous class to nested class

Transforming anonymous class to the nested one can be useful if the user decides to have more instances of this class. Let's have the *SuperClass* class and an anonymous class extending it:

```
public abstract static class SuperClass {
  public  SuperClass( int i) {
  }

  public abstract  String method();
}
```

```
 public static  void main( String[] args) {
  final String s = "string";
  SuperClass var = new SuperClass(9){
    @Override
    public  String method() {
           return s;
    }
  };
}
```

Now the user wants to transform anonymous class to a nested one – the final variable *s* will be passed through the constructor:

```
public static  void main( String[] args) {
  final String s = "string";
  SuperClass var = new NewClass(9, s);
}
```

```
private static class NewClass extends SuperClass {
  @Override
  public String method() {
     return s;
  }

  private String s;
  public  NewClass( int par1,  String s) {
     super(par1);
     this.s = s;
  }
}
```

Implementation

A new nested class declaration is created in the proper parent class. The class must be declared as static if the anonymous class has been part of a static method. If the anonymous class is an implementation of an interface, the nested class must implement it too (similar for extending an existing class). We have to be able to determine if the anonymous class implements an interface or extends a class – this information we get from appropriate attributes of the *anonymous-class* element.

A new field is created for each final variable that has been accessed within the anonymous class (and declared outside). The methods and their bodies can be just moved – we only need to replace references to the removed final variables by references to the newly created fields.

Constructor has two groups of parameters – the first group is passed to the super constructor and the other one is used to initialize the fields. The body of the constructor consists of calling the super constructor and initializing all fields that have replaced final variables.

6.1.8. Inline method

A method call is replaced by the body of the called method. Parameters of the method are replaced by local variables. Unfortunately not all methods can be inlined.

Inlining is available only for simple methods (consisting of a single return statement) that are used in assignment, simple condition or argument. Methods' invocations that are parts of more complicated expressions can't be inlined – for example:

int a = 3;

int b = ++a + foo(a);

In this case the invocation of the method *foo* cannot be inlined because it would change the result of the expression.

Obviously it's not possible to inline a method having no body (abstract method or method declared in an interface). The method that is being inlined must contain at most one return statement.

All fields, types and methods that are accessed from the body of the inlined method must be accessible from the position of the invocation that is being transformed.

Example:

Let the user has class *ClassB* and it's invocation from another class:

```
public class ClassB {
  protected int protectedInt = 5;
  public  void methodA( int a) {
     System.out.print(a);
      }

  public  String methodB( String s) {
     System.out.print(s + protectedInt);
     methodA(7);
     return null;
  }
}
```

```
public class TestClass {
public static  void main( String[] args) {
   …
  int s = 0
  new ClassB().methodB("string");
   …
  }
}
```

Inlining of the *methodB* will be successful only if both classes are in the same package (field *protectedInt* must be accessible). After the transformation the user gets:

```
public class TestClass {
  public static  void main( String[] args) {
      …
     int s = 0
     ClassB r = new ClassB();
     String _s = "string";
     System.out.print(s + r.protectedInt);
     r.methodA(7);
      …
  }
}
```

The parameter *s* has been changed to the local variable *_s* because the local variable *s* had already existed.

Sometimes the user doesn't know what method should be inlined:

```
public interface Interface {
  public  void methodA( int A);
}
```

Let's have classes A, B and C – all implementing the *Interface*. Now let the user have this part of method:

```
…
Interface var;
…… // initialize var
var.methodA(8);
…
```

```
public class A implements Interface {
  public int field = 3;
  public void methodA(int i) {
     System.out.print(field + i);
  }
}
```

The user wants to inline the *methodA* invocation but he doesn't know what implementation of this method will be called in runtime. He can use the inline transformation without specifying what implementation should be used. This transformation will inline all possible implementations and uses the *instanceof* test to determine which one will be used. The transformation of the example will look like this:

```
…
Interface var;
…… // initialize var
Interface r = var;
if (r instanceof A) {
  System.out.print( ((A) r).field + I );
} else if (r instanceof B) {
  // body of methodA implemented in class B
} else {
  // body of methodA implemented in class C
}
…
```

Implementation

The user can enter one or two parameters. The first one is a method invocation that should be inlined. The second one (optional) is a method implementation that should be used for the inlining – if the user doesn't enter this parameter, all implementations will be inlined and the right one will be chosen in runtime using the *instanceof* test.

At first all conditions that we have mentioned recently must be checked. The most complicated is to check the accessibility of all fields, types and methods referenced within the method's body – for this purpose we have the method *canAcces* in the *TreeInfo* class (see Javadoc).

The visitor pattern is used to go through all fields, types and methods referenced within the method. It focuses on the members of the class that contains the method that is being inlined (*ClassB* in our example). All references to these members must be replaced by the references to the members of the newly created *ClassB* instance (in our example the variable *r*).

The same instance has to be used to access the class members –

```
new ClassB().methodA(7);
new ClassB().protectedInt;
```

can't be used because it can change the result. A new local variable must be declared:

```
ClassB r = new ClassB();
r.methodA(7);
r.protectedInt;
```

If there is at most one member accessed, this variable doesn't have to be created.

The next step is to replace method's arguments by local variables and change all references to the arguments by the references to these variables. No variable with the same name can be already declared (if it is, "_" is added in front of the new variable's name).

If the method invocation has been part of an expression, it will be replaced by the return value of the method. The last step is to fix imports using the *getNeededImports* method of the *TreeInfo* instance.

The transformation that inlines all possible implementations must also generate switch (using if – else statements with *instanceof* conditions) to choose the right implementation in runtime.

6.1.9. Introduce variable

Sometimes the user would like to replace a complicated expression by a reference to a local variable that contains this expression. Let's have this example:

```
int a, i;
double d;
…
int b = a * d + i – 5;
```

It would be useful to replace "*a * d + i*" by a single variable reference – it can make the source code more transparent. The user will get:

```
…
double newVar = a * d + I;
int b = newVar – 5;
```

The introduced variable can be optionally declared as final. It is also possible to replace all occurrences of the same expression.

Implementation

Introducing a single occurrence of an expression isn't difficult – new variable's declaration must be added and initialized by the expression that is being transformed. The old expression is replaced by a reference to this variable. The type of the variable is found by calling the *getType* method of the *TreeUtils* class (see the Javadoc).

If all occurrences of the expression should be replaced, it is necessary to find the right place where the new variable should be declared. This place can be easily found by calling the *getNearestCommonAncestor* method of the *TreeUtils* class (see the Javadoc).

6.2. Predicate abstraction

The predicate abstraction transformation is another example demonstrating capabilities of the XMLJT framework. The goal of this transformation is to convert a common Java program to a Boolean program (program having just boolean variables). The Boolean programs have smaller state space (except multi-thread programs and programs with recursion) so it's possible to use the Boolean program model checker to ensure the program is safe (there is no reachable error state). You can find information how to run the transformation in the User manual (Appendix B.6.11).

Prerequisites and requirements

This implementation of the predicate abstraction transformation should be just a demonstration of possibility to create such a difficult transformation using the XMLJT framework. That's why several prerequisites have to be satisfied.

- Predicates – It's very hard problem to find the right predicates automatically so the transformation has to get these predicates in its input.
- Single method program - Calling of methods is never transformed – result of calling the method is replaced by calling the user defined random function. All programs to be transformed should have only one method (we can satisfy this condition using the inlining transformation).
- No *for* and *switch-case* statements – These statements are not supported because they are meaningless in Boolean programs. All other control flow statements can be used without any restrictions.
- Integer variables – All variables should be integer because the theorem prover accepts only integers. Variables of other types will be removed and will have no effect on the Boolean program.

Implementation

Predicate abstraction transformation has three input parameters:

- method to be transformed
- list of predicates
- initial values of these predicates (mostly it would be possible to get this information from the source code using the theorem prover but there are some situations when it's difficult – for example if the integer variable is declared later in the method's body – like: int i = 0; i = i + 3; int k = i + 9; having predicate k > 10)

The user also must define (configuration):

- path to the theorem prover
- format of the theorem prover arguments
- name of the random function (returning random boolean value)

Let's have this input example to demonstrate implementation of the predicate abstraction transformation:

Method to be transformed:

```
public static void main(String[] args) {
  int i = 0;
  int x = 1;
  int k = 0;

  x = k +3;
  while (i < 100) {
     x += i;
     i++;
  }
  assert (x > 0);
}
```

Predicates:

p0: i < 0 (initial value is false)

p1: x > 0 (initial value is true)

Now let's go through the steps of the implementation:

1. Create boolean variables

Local variable of boolean type must be declared for each predicate. Initial value of these variables is set on the basis of the input parameters of the transformation. Let's see how our example has changed (bold typed statements have been already transformed):

```
public static void main(String[] args) {
  boolean p0 = false;
  boolean p1 = true;
  int i = 0;
  int x = 1;
  int k = 0;

  x = k +3;
  while (i < 100) {
     x += i;
     i++;
  }
  assert (x > 0);
}
```

2. Replace the predicates' occurrences

A simple visitor is used to remove all conditions that are equal to some predicate and replace it by the reference to the variable representing this predicate.

```
public static void main(String[] args) {
  boolean p0 = false;
  boolean p1 = true;
  int i = 0;
  int x = 1;
  int k = 0;

  x = k +3;
  while (i < 100) {
     x += i;
     i++;
  }
  assert (p1);
}
```

3. Transform all statements to the Boolean program

The visitor pattern is used to go through all statements and transform them to the Boolean program statements – let's describe this transformation in three steps (see the Figure 3):

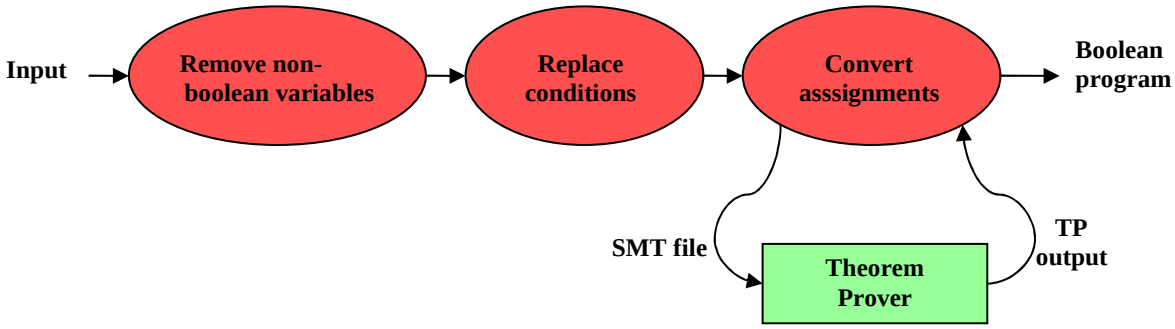


Figure 3: schema of the Boolean program creation

a. Remove declarations of non-boolean variables

All declarations of local variables must be removed – except local variables of the boolean type. However, references to these variables are still remaining in the method's body – at this moment the source code is not valid.

```
public static void main(String[] args) {
  boolean p0 = false;
  boolean p1 = true;

  x = k +3;
  while (i < 100) {
     x += i;
     i++;
  }
  assert (p1);
}
```

b. Replace other conditions

Conditions that aren't equal to any of predicates must be replaced by calling the user defined random function (in our example this function is called *myRandom*).

```
public static void main(String[] args) {
  boolean p0 = false;
  boolean p1 = true;

  x = k +3;
  while (myRandom()) {
     x += i;
     i++;
  }
  assert (p1);
}
```

c. Transform assignment expressions

There are three kinds of assignment expressions. Note: unary expressions (like i++) are treated like assignments (*i = i + 1*).

- All assignments with non-predicate variable (variable isn't contained in any predicate) on their left side can be ignored because they will have no effect on the result of the Boolean program. There is no such statement in our example – "k = i * 2" would be assignment of this type.
- Now we have only statements that can change value of at least one predicate. The next step is to transform all assignments that are independent on the predicates' values. It means their right side doesn't contain only predicate variables and constants (in our example "x = k + 3" – k isn't part of any predicate). In these cases it is not possible to calculate the result of the expression on the right side. In our example the variable *x* is contained only in one predicate so only one assignment will be generated (in general one assignment must be generated for each predicate containing variable that is on the right side of the origin assignment).

```
public static void main(String[] args) {
  boolean p0 = false;
  boolean p1 = true;

  p1 = myRandom();
  while (myRandom()) {
     x += i;
     i++;
  }
  assert (p1);
}
```

- The last step is to transform assignments that satisfy these conditions:
    a) the variable on the left side is part of at least one predicate
    b) the right side contains nothing but constants and references to the variables contained in at least one predicate

It's necessary to call the theorem prover to transform these statements. For each assignment new values of predicates must be set depending on their old value. However, not all predicates have to be changed and new values of changed predicates don't have to depend on all predicates. Let's have "*x =+ i;*" statement from our example (with the same predicates). Here are all possible predicates' values before and after statement:

Before		Assignment	After		SAT
x > 0 (p1)	**i < 0 (p0)**	**x1 == x + i**	**x1 > 0 (p1)**	**i1< 0 (p0)**	
true	true	true	true	true	?
true	true	true	true	false	?
true	true	true	false	true	?
true	true	true	false	false	?
true	false	true	true	true	?
true	false	true	true	false	?
true	false	true	false	true	?
true	false	true	false	false	?
false	true	true	true	true	?
false	true	true	true	false	?
false	true	true	false	true	?
false	true	true	false	false	?
false	false	true	true	true	?
false	false	true	true	false	?
false	false	true	false	true	?
false	false	true	false	false	?

Table 1: the predicates table before reducing

This table can be very large if we have many input predicates – exact size of the table will be *2^(2*p)* where *p* is the number of predicates. Usually it's possible to rapidly reduce the size of this table. In the "before" part of the table there will be only predicates that contain at least one variable referenced on the right side of the assignment. In the example no predicate can be removed. In the "after" part of table there will be only predicates that contain variable from the left side of the assignment. In the example only "x1>0" will stay. The table after reducing:

Before		Assignment	After	SAT
x > 0 (p1)	**i < 0 (p0)**	**x1 == x + i**	**x1 > 0 (p1)**	
true	true	true	true	?
true	true	true	false	?
true	false	true	true	?
true	false	true	false	?
false	true	true	true	?
false	true	true	false	?
false	false	true	true	?
false	false	true	false	?

Table 2: the predicates table after reducing

Now the theorem prover must be called to determine which rows of the table can be satisfied. There are many theorem provers using their own native input format. Fortunately most of them accept also the *smt* [16] input format that is used for benchmark in some theorem provers' competitions. The smt file is generated for each row of the table

and it is passed to the theorem prover. The theorem prover returns “sat” (row can be satisfied) or “unsat” (row can never be satisfied) value. After calling the theorem prover for all rows of the table we get:

Before		**Assignment**	**After**	**SAT**
x > 0 (p1)	**i < 0 (p0)**	**x1 == x + i**	**x1 > 0 (p1)**	
true	true	true	true	**SAT**
true	true	true	false	**SAT**
true	false	true	true	**SAT**
true	false	true	false	**UNSAT**
false	true	true	true	**UNSAT**
false	true	true	false	**SAT**
false	false	true	true	**SAT**
false	false	true	false	**SAT**

Table 3: output of the theorem prover

On the basis of this table new switch statement will be generated (using if - else condition). Number of branches in this switch is equal to *2^b* where *b* is the number of predicates in the “before” part of the table (in our example *b*=2).

```
if (p1 && p0) {
  p1 = myRandom(); // we don’t know new value of p1
} else if (!p1 && p0) {
  p1 = false;
} else if (p1 && !p0) {
  p1 = true;
} else if (!p1 && !p0) {
  p1 = myRandom(); // we don’t know new value of
}
```

This switch will work but it can be simplified to be user friendly. The last branch doesn’t have to contain condition. Some branches have the same content so they can be represented by a single branch. Statement after simplifying:

```
if (!p1 && p0) {
  p1 = false;
} else if (p1 && !p0) {
  p1 = true;
} else {
  p1 = myRandom();
}
```

After going through all assignments the predicate abstraction transformation is finished. The final state of the example (compared to the origin):

BEFORE	AFTER

public static void main(String[] args) {
int i = 0;
int x = 1;
int k = 0;
 	
x = k +3;
while (i < 100) {
x += i;
i++;
}
assert (x > 0);
}
	public static void main(String[] args) {
	boolean p0 = false;
	boolean p1 = true;
	
	p1 = myRandom();
	while (myRandom()) {
	if (!p1 && p0){
	p1 = false;
	} else {
	if (p1 && !p0){
	p1 = true;
	} else {
	p1 = myRandom();
	}
	}
	if (p0){
	p0 = myRandom();
	} else {
	p0 = false;
	}
	}
	assert (p1);
	}

Table 4: changes in the source code

Chapter 7. Evaluation

After implementing several transformations the usability of the XMLJT framework can be evaluated.

There was no significant complication while implementing transformations from the Chapter 6 but the XML-based transformation framework has got some disadvantages, too.

Auto-generated JAXB classes of tree nodes are one of the biggest benefits of the XML technology. It wasn't necessary to write our own tree nodes classes so we avoided possible errors while implementing them. However, the JAXB isn't as powerful tool as we expected. It was necessary to create extensions for all generated classes so the framework will have to be changed every time the Java schema will change.

While using JAXB classes it's not possible to create invalid XML document so it's not necessary to validate XML output of transformations. Of course, it is possible to create a valid XML document that represents an invalid Java source code (having compilation errors).

XML technology also enabled to define references between tree elements. These references help to implement effective transformations. There is a reference table that enables fast searching of tree nodes by their ids.

The XSLT technology is a parallel transformation framework that we have got for free (just for using the XML for representing the Java source code). This technology can be used for performing some simple transformations.

Implementation of the predicate abstraction transformation (the Chapter 6.2) showed that it's quite easy to cooperate with external tools (integration of the theorem prover).

XML-based approach to transformation frameworks seams to be useful also for other programming languages, especially for the similar ones (like C#).

Chapter 8. Conclusions

8.1. Conclusion

The XMLJT transformation framework has been completed and it is now ready to be used.

XML Schema suitable for representation of Java sources has been defined by modification of an existing schema (JavaML). Parsing of Java sources is ensured by the Recoder tool. The JAXB technology is used to parse XML files (creating in-memory representation) and generate XML files on the basis of the in-memory source tree. It also automatically (according to the XML Schema) generates classes for in-memory representation of the source tree. Transformations and analyses are able to easily obtain information about references within the source tree. All nodes within the tree can be accessed directly using the reference table.

Several transformations have been implemented to proof the usability of the XMLJT framework. Especially non-trivial transformations (like inlining and predicate abstraction) show how simple is to implement transformations and analyses using the XMLJT framework and also demonstrate possibility of cooperation with external tools.

It has been proved that the XML-based approach to developing transformation frameworks does work. Of course, there are some advantages and also some disadvantages of this approach (see the Chapter 7).

8.2. Future work

The core of the framework won't need any discontinuous innovations until the new version of the Java programming language (now the Java 1.5 is supported).

In the future the framework may reflect also comments and formatting of the source code. Now if the user converts Java files to the XML and back to the Java he will lose all comments and information about source code formatting. This fact can be uncomfortable if the user is going to work with the source code after performing the transformation.

To reflect formatting of the source code the XML files have to contain information about formatting and comments. However, it must not rapidly increase the size of the XML files and also the size of the in-memory representation of the source tree.

Some sample transformations that are part of the framework can be improved in the future. Especially the predicate abstraction transformation can be more sophisticated.

Cooperation with external tools can bring some additional information to the XMLJT framework. For example it’s possible to cooperate with the Soot framework to get control-flow graph or points-to analysis information.

Similar transformation frameworks for other programming languages (e.g. C# or C++) can be also developed on the basis of the XMLJT in the future.

References

[1] Ademar Aguiar, Gabriel David, and Greg Badros, "JavaML 2.0: Enriching the Markup Language for Java Source Code", XATA'2004.

[2] Bill Evjen, Kent Sharkey, Thiru Thangarathinam, and Michael Kay, Professional XML, Wiley Publishing, 2007

[3] Brett McLaughlin and Justin Edelson, Java and XML, O'Reily Media 2007

[4] http://recoder.sourceforge.net/wiki/

[5] http://www.antlr.org/

[6] https://javacc.dev.java.net/

[7] http://www.eclipse.org/jdt/overview.php

[8] http://platform.netbeans.org/tutorials/60/nbm-copyfqn.html

[9] http://www.eclipse.org/

[10] http://www.netbeans.org/

[11] Raja Vallée-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co, Soot - a Java Optimization Framework, September 1999

[12] http://cil.sourceforge.net/

[13] Brian J. Gough: An Introduction to GCC, Network Theory Ltd., 2005

[14] Marc Loy, Robert Eckstein, David Wood, James Elliott, and Brian Cole, Java Swing, Second Edition, O'Reily Media 2003

[15] http://code.google.com/p/jsyntaxpane/

[16] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, Helmut Veith, Counterexample-Guided Abstraction Refinement, CAV 2000: 154-169

[17] Silvio Ranise, and Cesare Tinelli, The SMT-LIB Format: An Initial Proposal, PDPAR'03

Appendix A. CD-ROM content

Included CD-ROM contains these directories and files:

- documents – text documents describing the project
  - javadoc – Javadoc generated documentation of the framework classes
  - thesis.pdf – text of this thesis in pdf format

- install – ready to run the framework and proof of the concept transformations
  - XMJT.jar – core of the framework – this library should be used by the user to write his own transformations and analyses
  - refactoring.jar – implementation of refactoring transformations
  - predicate-abstraction.jar – implementation of predicate abstraction transformation
  - XMLJTgui
    - XMJTgui.jar – executable jar of the XMLJT GUI
    - lib – libraries that the GUI needs

- sources – source files of the project
  - XMLJT – sources of the main project (XMLJT framework)
    - jaxbplugin – sources of a simple JAXB plugin
    - lib – libraries used by the framework
    - src – Java source files of the framework
    - test – unit tests and test data
    - xml-resources – XML Schema and JAXB binding file
      - java.xsd – XML Schema of the Java 1.5
      - bind.xjb.xml – JAXB binding file
    - build.xml – Ant build script
  - XMLJTgui – source of the XMJTgui
    - lib – libraries used by the GUI
    - src – Java source files of the GUI
    - build.xml – Ant build script

Appendix B. User Manual

B.1. Requirements

The XMLJT framework is platform independent. It has been successfully tested in Windows and Linux operation systems (including XMLJT GUI). Memory requirements depend on the size of transformed project. 1 GB memory should be enough even for very large projects.

B.2. Installation

Install compiled jar files

1) download compiled files from the website http://xmljt.sourceforge.net or from the *install* directory of the CD-ROM
2) decompress the files (only when downloading from the website)
3) now you can use the framework as a jar library, perform the transformations from the command line or launch the graphic user interface

Compiling and testing sources

If you want to build jar files from the sources you must have the Ant tool installed – you can download it from http://ant.apache.org for free.

1) download source files from the website http://xmljt.sourceforge.net or from the *sources* directory of the CD-ROM
2) decompress files (only when downloading from the website)
3) to build the XMLJT framework go to the *XMLJT* directory and run Ant (use "ant" command) – jar files will be created in the *dist* subdirectory
4) to build the XMLJT GUI go to the *XMLJTgui* directory and run Ant (use "ant" command) – jar file will be created in the *dist* subdirectory
5) if you want to test the framework (for example after modifying its sources) you can use unit tests that are part of the sources – to run the tests go to the *XMLJT* directory and type "ant test" (predicate abstraction test needs a theorem prover – see the Appendix B.6.11 for information how to obtain it, then you have to set the theorem prover path in the test source file)

Note: if you want to compile the sources using some SDK (like NetBeans or Eclipse) you must also add automatically generated files (these files are generated using the JAXB tool)

– these files can be found in the *build/generated/jaxbCache/jaxb* directory (you must run Ant in the *XMLJT* directory to generate these files)

B.3. Conversion from Java to XML

There are three ways how to convert Java source files to a single XML file:

- **from the command line**

Executable class *xmljt.java2xml.JavaToXml* can be used to convert Java files to XML from the command line. If you need to pass some additional classpath that is necessary for parsing the Java files you can pass it to the JVM (using –cp option).

Parameters:

if you want to transform all Java files in a directory (including its subdirectories):

<input directory> <output file>

if you want to transform separate Java files:

<input file> ... <input file> <output file>

Example:

java -cp XMLJT.jar xmljt.java2xml.JavaToXml inputDirectory output.xml

- **conversion in your own source code**

There are two static methods called *convert* in the *xmljt.java2xml.JavaToXml* class. The only difference is the format of input Java files. The first one converts set of single files. The other one converts all files in given directory (and its subdirectories).

The second parameter of both methods is the name of output XML file. The last parameter is optional and defines the classpath to be used while parsing the Java files.

```
String error = JavaToXml.convert(sourceDir, xmlFile);
if (error != null) {
   System.err.print("Error while parsing java files: " + error);
}
```

- **transform the files using XMLJT GUI**

It's also possible to use the XMLJT graphic user interface to convert Java files to a single XML file. Use the "Import Java files (directory)" item of the main menu (for further information see the Appendix B.5).

Note: *OutOfMemoryError* exception can be thrown while parsing large projects – increasing of the JVM memory (using Java command options "-Xms" and "-Xmx") will fix it.

B.4. Output to Java

There are three ways how to generate Java source files from the in-memory representation of the program.

- **from the command line**

Executable class *xmljt.transformations.tojava.ToJava* can be used to convert Java files to XML from the command line.

Parameters:

The transformation has two parameters: name of XML file to be transformed and name of directory where the Java sources should be generated.

Example:

java -cp XMLJT.jar **xmljt.transformations.tojava.ToJava** input.xml outputDir

- **in your own source code**

There is a static method called transform in the *xmljt.transformations.tojava.ToJava* class. This method has two parameters: root of the source tree to be exported and name of the directory where the Java sources should be generated.

If you need to export just one file you can use the *getJavaFileSource* method in the same class.

- **Using the XMLJT GUI**

It's also possible to use the XMLJT graphic user interface to generate Java files. Use the "Export to Java Files…" item of the main menu (for further information see the Appendix B.5).

B.5. XMLJT GUI

Here is a short description of the XMLJT graphic user interface. To launch the XMLJT GUI go to the XMLJTgui directory and run the *XMLJTgui.jar* file (use "java -jar XMLJTgui.jar" command). Java Runtime Environment (JRE) 1.6 must be installed to run the GUI (you can download it from http://java.sun.com/javase/downloads/index.jsp for free).

Main window

The main window of the XMLJT GUI has two parts (see the Figure 4). On the left side there is a tree of the project – Java files are placed in the directory defined by their package. The user can open a directory or a Java file using a mouse double-click. When the user opens a Java file he can see its content in the right part of the main window.

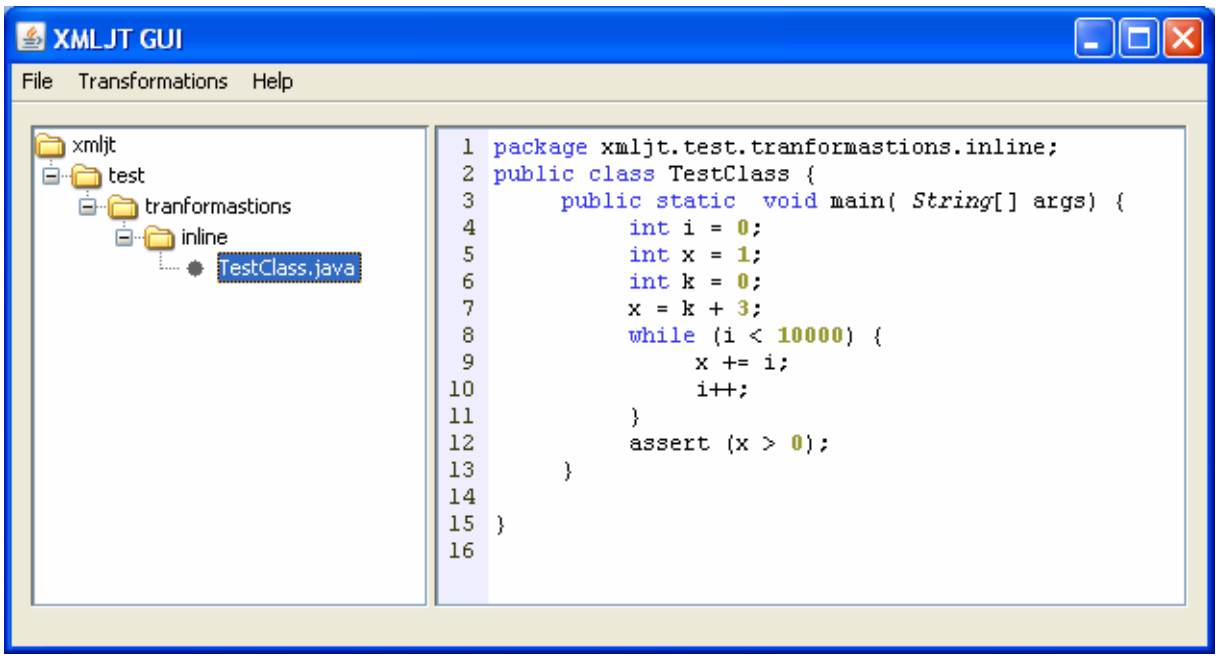


Figure 4: XMLJT GUI main window

Working with files and projects

There are two basic ways how to open a project. The user can either open an existing XML project file (using the "Open XML…" command of the main menu or Ctrl+O shortcut) or create a new XML project by importing Java sources.

Ctrl+Alt+I shortcut or the "Import Java Files…" command of the main menu can be used to import separate Java source files. List of imported files is set using the modal dialog (Figure 5). The user must also define name of XML file where the XML project should be stored. If the imported sources use some libraries not included in the current classpath the user must also define them in the dialog.

It's also possible to import whole directories – all Java files in the directory and its subdirectories will be imported. The user can use the "Import Java Directory…" command of the main menu or Ctrl+I shortcut. Displayed dialog (Figure 6) is similar to the previous one (importing separate Java files). The only difference is the user can choose only one directory instead of the list of files.

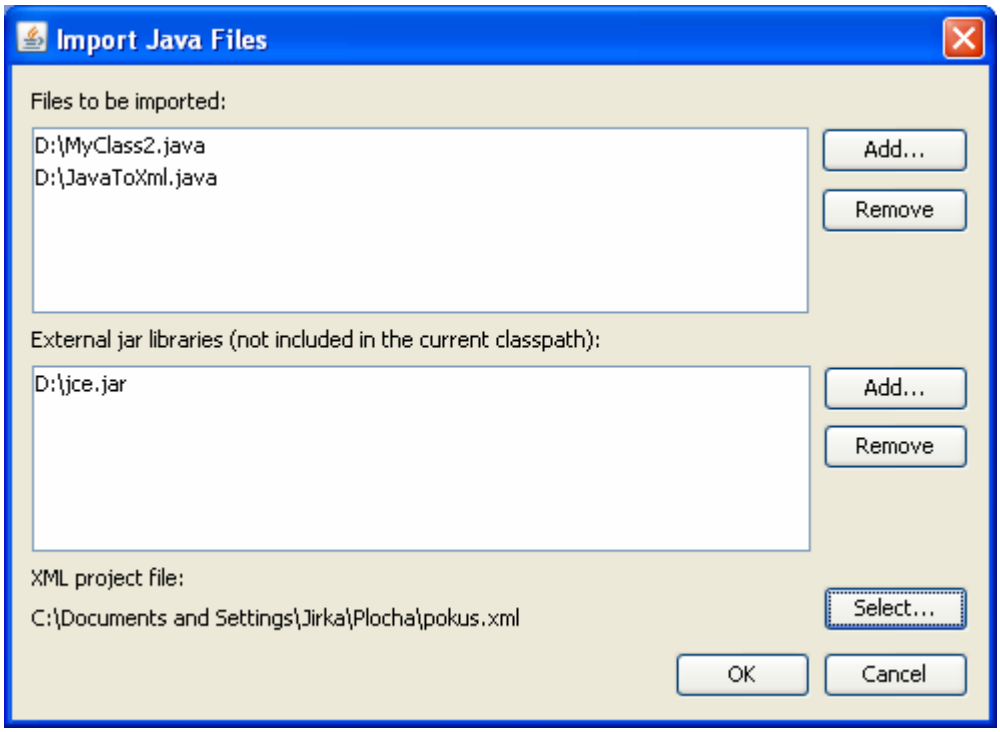


Figure 5: importing separate Java files

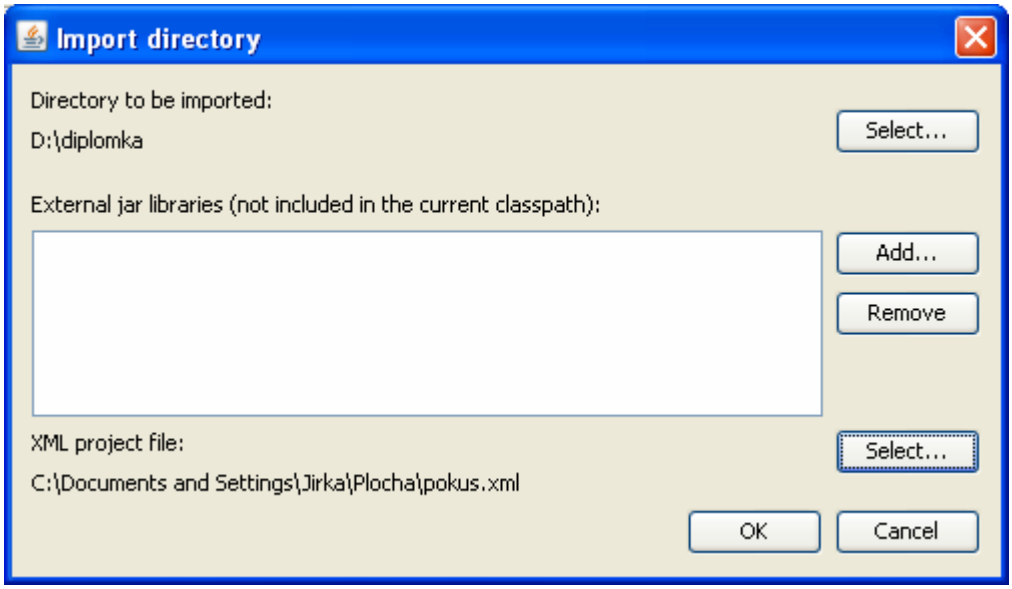


Figure 6: importing whole directory

Once the user has finished the transformations he can save the XML project using the "Save XML" command of the main menu or Ctrl+S shortcut. If he clicks on the "Save XML as…" item of the main menu he can save new copy of the project using different file name.

For compiling the program it's necessary to export the project back to the Java sources. The user can use either Ctrl+E shortcut or the "Export to Java Files…" item of the main menu. Then he must choose the directory where the sources should be exported. The directory tree corresponding to the structure of packages will be created in the target directory.

Running the transformations

There are two ways how to perform transformations. The first one is to use the main menu ("Transformations") and click on the name of transformation to be performed. The second way is to use the pop-up menu (right mouse click anywhere in the source code) and its "Transformations" submenu.

Some transformations can be used only if some specific part of the source code is selected – for details see the description of the specific transformation (B.6). It's possible to add new transformations to the GUI dynamically. See the tutorial in the Appendix C.3 for details.

B.6. Performing transformations

B.6.1. Refactoring in general

There are three ways how to run refactoring transformations.

- **from the command line**

To run the transformations from the command line go to the folder containing *refactoring.jar* file and run **"java -jar refactoring.jar"** having these parameters:

-help	print help information
-xi	xml input file (required) – if you want to create xml project file from Java sources, see the Appendix B.3
-xo	xml output file (optional)
-t	name of transformation to be performed and its parameters – you can find detailed information about parameters in the description of the corresponding transformation

- **invoke the transformation from your own source code**

You will need *XMLJT.jar* and *refactoring.jar* libraries to run the refactoring transformations. At first you have to create new instance of the *transformations.refactoring.Refactor* class. *TreeInfo* object is the only parameter of the constructor.

Now you can call methods of the *Refactor* class to perform transformations (information about these methods can be found in the description of the corresponding transformation).

- **run transformation using the XMLJT GUI**

It's also possible to use the graphic user interface to run all refactoring transformations. See the description of the corresponding transformation for details.

B.6.2. Method extracting

- **from the command line**

Name of transformation:

extractMethod

Parameters:

-name name of the method to be created (required)

-stmts ids of the statements to be extracted (required)

-mod access modifier of the method to be created (optional – if there is no such argument the default access modifier will be used)

Example:

java -jar refactoring.jar -xi input.xml **-xo** output.xml **-t** extractMethod **-mod** public **-stmts** -35 -42 **-name** newMethod

- **invoke the transformation from your own source code**

The *extractMethod* method has three parameters: name of method to be created, access modifier of this method and list of ids of statements to be extracted.

```
Refactor r = new Refactor(treeInfo);
r.extractMethod("newMethod", ModifierType.PUBLIC, -35, -42);
```

- **run transformation using the XMLJT GUI**

Select statements to be extracted and choose item “Introduce Method” from the main menu or the pop-up menu. Then you have to type name of the method and choose its modifier in a simple dialog.

B.6.3. Renaming

- **from the command line**

Name of transformation:

rename

Parameters:

There are only two parameters: id of element to be renamed and new name.

Example:

java -jar refactoring.jar -xi input.xml **-xo** output.xml **-t** rename 116 newName

- **invoke the transformation from your own source code**

There are many rename methods – one for each type of element that can be renamed. All of them have two parameters: element to be renamed and its new name.

```
Refactor r = new Refactor(treeInfo);
r.rename(element, “newName”);
```

- **run transformation using the XMLJT GUI**

Select statement to be renamed and choose item “Rename” from the main menu or the pop-up menu. Then you have to type new name in a simple dialog.

B.6.4. Change method parameters

- **invoke the transformation from your own source code**

There are three methods for manipulating with method’s parameters. New parameter can be added by calling *addParameter* method. It has got three parameters: method whose declaration should be changed, new parameter to be added (instance of the *FomalArgument* class) and default value of the new parameter (any implementation of the *Expression* interface, will be used every time the transformed method is being called).

The method *removeParameter* is used to remove parameter. It has two parameters: method to be changed and name of the parameter to be removed.

The user can also change order of method’s parameters using the *permuteParameters* method. It has got two parameters: method to be changed and array of integer that defines the permutation of the parameters. For example let’s have a method with parameters (a, b, c) and transform it with {3, 1, 2} permutation of parameters. The transformed method will have parameters (c, a, b) – order of parameters will be also changed everywhere the method is being called.

```
Refactor r = new Refactor(treeInfo);
Type type = factory.createType(TreeUtils.INT_TYPE_NAME, TreeUtils.INT_TYPE_ID);
FormalArgument newParam = factory.createFormalArgument(type, "newParam", false);

// add int parameter having default value 0
r.addParameter(method, newParam, factory.createLiteralNumber(0));
// remove just added parameter
r.removeParameter(method, "newParam"));
// switch two parameters of method
r.permuteParameters(method, new int[]{2,1}));
```

- **run transformation using the XMLJT GUI**

Select the method (or any of its invocations) to be changed and choose the item "Change Method Parameters" from the main menu or the pop-up menu. You can easily change parameters using displayed dialog.

B.6.5. Variable to field

- **from the command line**

Name of transformation:

localToField

Parameters:

-name	name of the field to be created (required)
-var	id of the local variable to be transformed (required)
-mod	modifiers of the field to be created (optional – if there is no such argument, the default modifier will be used)

Example:

java -jar refactoring.jar -xi input.xml **-xo** output.xml **-t** localToField **-name** newField **-var** 116 **-mod** public

- **invoke the transformation from your own source code**

The *variableToField* method of the *Refactor* class has three parameters: local variable to be transformed, name of the field to be created and its modifiers.

```
Refactor r = new Refactor(treeInfo);
r.variableToField(localVariable, "newField", modifiers));
```

- **run transformation using the XMLJT GUI**

Select local variable to be changed and choose the item "Variable to Field" from the main menu or the pop-up menu. Then you have to type name of the field and choose its modifiers in a simple dialog.

B.6.6. Extract interface

- **from the command line**

Name of transformation:

interface

Parameters:

-package	name of a package where the new interface should be created (required)
-name	name of the interface to be created (required)
-class	id of the class to be transformed (required)
-methods	ids of methods that should be declared in the interface (required)

Example:

java -jar refactoring.jar -xi input.xml **-xo** output.xml **-t** interface **-package** my.package **-name** NewInterface **-class** 30 **-methods** 175 200

- **invoke the transformation from your own source code**

The *extractInterface* method of the *Refactor* class has four parameters: name of the interface, its package, the class to be transformed and list of the methods to be declared in the interface (vararg).

```
Refactor r = new Refactor(treeInfo);
r.extractInterface(“my.package”, “NewInterface”, testClass, methodA, methodB));
```

- **run transformation using the XMLJT GUI**

Select the class to be changed and choose the item “Extract Interface” from the main menu or the pop-up menu. Then you use displayed dialog to choose methods that should be declared in the interface.

B.6.7. Extract super class

- **from the command line**

Name of transformation:

super

Parameters:

-package	name of a package where a new super class should be created (required)
-name	name of the super class to be created (required)
-class	id of the class to be transformed (required)
-methods	ids of the methods that should be declared and implemented in the super class (optional)
-abstract	ids of the methods that should be declared as abstract in the super class (optional)

Example:

java -jar refactoring.jar -xi input.xml **-xo** output.xml **-t** super **-package** my.package **-name** NewClass **-class** 30 **-methods** 175 200 **-abstract** 244

- **invoke the transformation from your own source code**

The *extractSuperClass* method of the *Refactor* class has five parameters: name of the super class, its package, the class to be transformed, list of the methods to be moved to the super class (including their body) and list of the methods that should be declared as abstract in the super class.

```
Refactor r = new Refactor(treeInfo);
r.extractSuperClass("my.package", "NewClass", testClass,
         new Method[]{methodMain}, new Method[0]));
```

- **run transformation using the XMLJT GUI**

Select the class to be changed and choose the item "Extract Superclass" from the main menu or the pop-up menu. Then you use the displayed dialog to choose the methods that should be moved to the super class (or declared as abstract).

B.6.8. Anonymous class to nested class

- **from the command line**

Name of transformation:

anonToNested

Parameters:

-name name of the nested class to be created (required)

-anonymous id of the anonymous class to be transformed (required)

-mod modifiers of the class to be created (optional – if there is no such argument, the default modifier will be used)

Example:

java -jar refactoring.jar -xi input.xml **-xo** output.xml **-t** anonToNested **-anonymous** 441 **-name** NewNestedClass **-mod** protected

- **invoke the transformation from your own source code**

You can use the *anonymousToNested* method of the *Refactor* class that has three parameters: an anonymous class to be transformed, the name of a new nested class to be created and its modifiers.

```
Refactor r = new Refactor(treeInfo);
r.anonymousToNested(anonymousClass, "NewNestedClass", modifiers));
```

- **run transformation using the XMLJT GUI**

Select the anonymous class to be transformed and choose the item “Anonymous Class to Nested” from the main menu or the pop-up menu. You must define name of nested class and its modifiers in the dialog that will be displayed.

B.6.9. Inline method

- **from the command line**

Name of transformation:

inline

Parameters:

There is only one parameter: id of method’s invocation (xml element *method-call*) that should be inlined. All possible implementations will be inlined and switch (using if – else statements with *instanceof* conditions) will be generated to choose the right implementation in runtime.

Example:

java -jar refactoring.jar -xi input.xml **-xo** output.xml **-t** inline -42

- **invoke the transformation from your own source code**

There are two methods *inline* in the *Refactor* class. The first one has two parameters: method's invocation that should be inlined (instance of the *xmljt.xmltree.nodes.MethodCall* class) and the method whose body should be used for the inlining (the user is sure what implementation of the method will be used in runtime).

The other one has only one parameter – instance of the *xmljt.xmltree.nodes.MethodCall* class. In this case all possible implementations will be inlined and switch (using if – else statements with *instanceof* conditions) will be generated to choose the right implementation in runtime.

```
Refactor r = new Refactor(treeInfo);
r.inline(methodCallToBeInlined);
r.inline(otherMethodCall, methodToBeUsed);
```

- **run transformation using the XMLJT GUI**

Select the method's invocation to be inlined and choose the item "Inline" from the main menu or the pop-up menu. In displayed dialog you must choose which of possible implementations would you like to use.

B.6.10. Introduce variable

- **from the command line**

Name of transformation:

extractVar

Parameters:

-name	name of the variable to be created (required)
-expr	id of the expression to be transformed (required)
-final	the variable will be declared as final (optional)
-all	all occurrences of the expression will be replaced (optional)

Example:

java -jar refactoring.jar -xi input.xml **-xo** output.xml **-t** extractVar **-expr** -143
-name newVar **-final**

- **invoke the transformation from your own source code**

There are two similar methods. The first one is named *extractVariable* and replaces only the specified expression by the local variable (not all occurrences of the expression). It has three parameters: name of the variable to be created, boolean parameter specifying if the variable should be declared as final (true if we want it to be final) and the expression to be transformed (any implementation of the *xmljt.xmltree.nodes.Expression* interface).

The second method is named *extractVariableAllOccurences* and has the same parameters as the previous one. The only difference is that it replaces all occurrences of given expression.

```
Refactor r = new Refactor(treeInfo);
r.extractVariable(“newVar1”, true, expression1));
r.extractVariableAllOccurences(“newVar2”, false, expression2));
```

- **run transformation using the XMLJT GUI**

At first you have to select the expression you want to be stored in the local variable. Then choose the item “Introduce Variable” from the main menu or the pop-up menu and fill out the name of the newly created variable.

B.6.11. Predicate abstraction

You will need a theorem prover to run the predicate abstraction transformation. You can use any theorem prover that supports *smt* format. The transformation has been tested using two theorem provers: yices (you can download it for free from http://yices.csl.sri.com/) and z3 (you can download it for free from http://research.microsoft.com/en-us/um/redmond/projects/z3/).

- **from the command line**

To run the transformation from the command line go to the folder containing *predicate-abstraction.jar* file and type the following: **java -jar predicate-abstraction.jar**

List of parameters:

-help	print help information
-ji	java input file (optional)
-jo	java output directory (optional)
-xi	xml input file (required) - this file will be generated if you use -ji parameter
-xo	xml output file (optional)
-pred	list of predicates and its default values (required)
-tppath	path to the theorem prover executable file (required)
-tpargs	format of the theorem prover's arguments (optional - default value is "<file>"), “<file>” will be replaced by the name of smt input file
-random	name of static random function to be called (must return boolean; required)

Here are some examples:

java -jar predicate-abstraction.jar -ji TestClass.java **-xi** TestClass.xml **-xo** output.xml **-jo** output **-pred** "x>0" true "i<0" false **-tppath** D:\yices\bin\ yices.exe **-tpargs** "-smt <file>" **-random** myRandom

java -jar predicate-abstraction.jar -xi TestClass.xml **-jo** output **-pred** "x>0" true **-tppath** D:\z3\bin\z3.exe **-random** myRandom

- **invoke the transformation from your own source code**

You will need *XMLJT.jar* and *predicate-abstraction.jar* libraries to run the predicate abstraction transformation. At first you have to create new instance of the *xmljt.transformations.pa.PredicateAbstraction* class. TreeInfo object is the only parameter of the constructor.

Now you can call the method *transform* having three parameters:

- method to be transformed (program should have only one method)
- list of predicates (instances of the *xmljt.xmltree.nodes.Expression*) – if you have just string representation of the predicates you can use the *xmljt.transformations.pa.PredicatesParser* class to parse them (see the Javadoc for further information)

- list of initial values of the predicates (size of the list must be equal to the size of the list of predicates)

The *transform* method returns true if the transformation has been successful, false otherwise. Here is an example of a source code fragment invoking the predicate abstraction transformation:

```
PredicateAbstraction predicateAbstraction = new PredicateAbstraction(treeInfo);

boolean result = predicateAbstraction
                        .transform(method, parsedPredicates, initValues);
```

- **run transformation using the XMLJT GUI**

It's also possible to use the graphic user interface to run the predicate abstraction transformation. At first you have to open (import) xml project (see the Appendix B.5). Now you must select the method to be transformed (by setting the cursor position to its name). You can use either the pop-up menu (Transformations → Predicate Abstraction) or the main menu (Transformations → Predicate Abstraction) to open the setting dialog (Figure 7).

The most important is to define the set of predicates to be used – you must type an attribute to the field (1) and define its initial value (2). You can also see the list of already added predicates (3). It's also necessary to define path to the executable file of the theorem prover (4) and the format of the theorem prover's command line parameters (5). After typing the name of static random function (6) you can press the OK button to process the transformation.

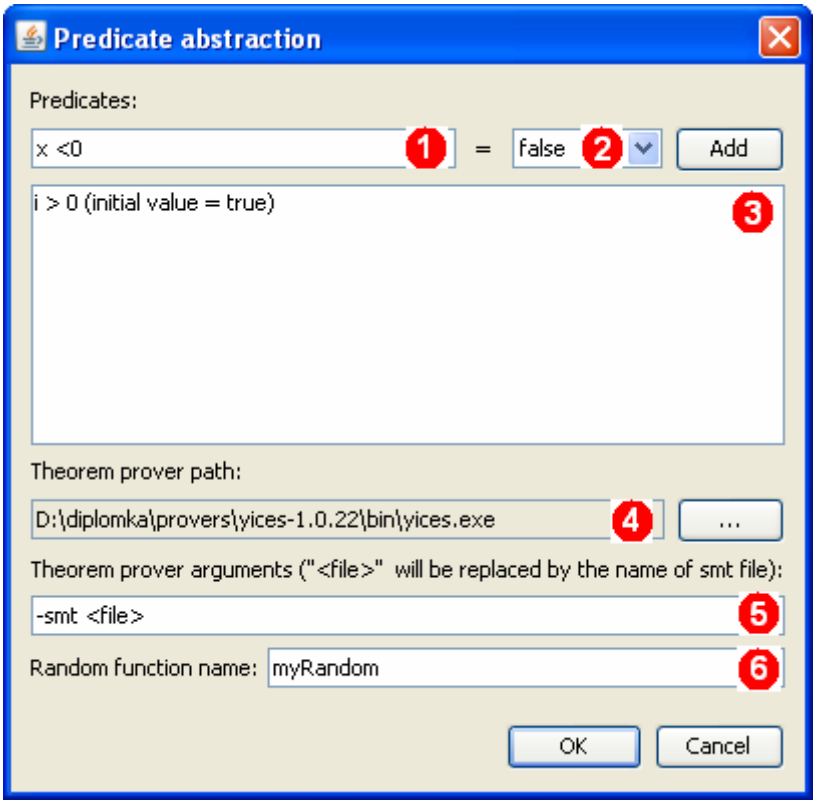


Figure 7: the predicate abstraction setting dialog

Appendix C. Tutorials

In the next chapter you can find three tutorials that should help the user to learn how to work with the XMLJT framework. All classes and methods used in these tutorials are described in the Javadoc, so they won't be described in full detail within the tutorials. Source code of the first two tutorials can be found in the *xmljt.transformations.tutorial* package.

C.1. Tutorial 1 – creating the program elements

This tutorial describes how to create a simple "Hello World" program using just the XMLJT framework. The framework is not the best way how to create Java programs but this tutorial wants to demonstrate how new program elements can be created – that's what the user will need to implement transformations.

The goal of the tutorial is to create program like this one:

```
package hello.world;

public class Main {
  public static  void main( String[] args) {
     System.out.println("Hello World !!!");
  }
}
```

Program tree

The first step is to create a new tree representing the Java program. Most of the users will never use this procedure because they will get the tree by parsing a XML file or Java files.

An empty tree is created by calling the static method *createEmptyTree()* of the *TreeInfo* class (1). It returns new instance of the *TreeInfo* class – this class holds all information about the tree. Its method *getRoot()* is used to get the root element of the tree (instance of the *JavaSourceProgram* class) (2). The factory class (instance of the *ProgramElementFactory* class) is needed to create new program elements – *getProgramElementFactory()* method of the *TreeInfo* class is used to get it (3).

```
 1  TreeInfo treeInfo = TreeUtils.createEmptyTree();
 2  JavaSourceProgram root = treeInfo.getRoot();
 3  ProgramElementFactory factory = treeInfo.getProgramElementFactory();
```

Main class and its file

Now it's necessary to create the main class of the program. This class must be contained in its own Java file. At first the class's modifiers must be created (having just the *public* modifier) using the factory (4). After that we can create new class's declaration (named "HelloWorld") using these modifiers (5). New instance of the *JavaClassFile* class must be created – again using the corresponding factory method (name of package and main class must be specified in its parameters) (6). The Java file must be added to the root of the program tree (7).

```
 4  Modifiers classModifiers = Factory
         .createModifiers(factory.createModifier(ModifierType.PUBLIC));
 5  ClassDeclaration mainClass =
         factory.createClassDeclaration(classModifiers, "Main");
 6  JavaClassFile file = factory
         .createJavaClassFile("hello.world", mainClass);
 7  root.addJavaClassFile(file);
```

Main method

The main method has only one argument – *args* (one dimensional array of String). At first type of the argument must be defined. Two arguments must be passed to the factory method – name of the type (as we can see it in the source code – including brackets) and id of the type (in this case id of the standard String type) (8). In this case *dimensions* and *primitive* arguments must be also set because we are referencing the array type (9, 10). Finally it's possible to create an object representing the parameters of the main method (11).

```
 8  Type argsType = factory.createType("String[]", TreeUtils.STRING_TYPE_ID);
 9  argsType.setDimensions("1");
10  argsType.setPrimitive("false");
11  FormalArguments arguments = factory.createFormalArguments(
         factory.createFormalArgument(argsType, "args", false));
```

The return type and modifiers of the main method must be defined (12, 13). Now the method can be created (14) and a new block can be set to the method (15, 16). The method must be added to the main class (17).

```
12  Type returnType = factory.createType(TreeUtils.VOID_TYPE_NAME,
        TreeUtils.VOID_TYPE_ID);
13  Modifiers methodModifiers =
        factory.createModifiers(factory.createModifier(ModifierType.PUBLIC),
        factory.createModifier(ModifierType.STATIC));
14  Method mainMethod = factory.createMethod("main", returnType,
        methodModifiers, arguments);
15  Block block = factory.createBlock();
16  mainMethod.setBlock(block);
17  mainClass.addMethod(mainMethod);
```

Println statement

There are several prerequisites to call the *println* method – target of this method is the static field declared in the *java.lang.System* class. However, this type is declared out of the source code tree (in the jar library). Method *getOuterDeclaration(String)* of the *TreeInfo* class is used to get id's of outer declarations (18, 19, 20). Once ids of outer declarations are known, target of the method can be created in three steps: create type referencing to the *java.lang.System* class (21), create reference to the static *out* field (22, 23) and finally create target of the method call (24).

```
18  int printlnDecl = treeInfo.getOuterDeclaration
        ("java.io.PrintStream.println(java.lang.String)");
19  int systemDecl = treeInfo.getOuterDeclaration("java.lang.System");
20  int systemOutDecl = treeInfo.getOuterDeclaration("java.lang.System.out");
21  Type targetType = factory.createType("System", systemDecl);
22  FieldRef targetField = factory.createFieldRef("out",
        systemOutDecl);
23  targetField.setType(targetType);
24  Target callTarget = factory.createTarget(targetField);
```

The only argument of the *println* method call is the string literal having the "Hello World!!!" value (25). Now the call statement can be easily created (26) and add to the main method's block (27).

```
25  Arguments printlnArgs = factory.createArguments
        (factory.createLiteralString("Hello World!!!"));
26  MethodCall printlnCall = factory.createMethodCall(callTarget,
        printlnArgs, "println", printlnDecl);
27  block.addStatement(printlnCall);
```

The last step is to export the program – it's possible to export it into the single XML file that can be later used to easily load the program to the XMLJT framework. The second possibility is to create the Java source files. Note: the directory "c:\HelloWorld" must exist!

```
28  Marshal.marshall(root, "c:\\HelloWorld\\source.xml");
29  ToJava.transform(root, "c:\\HelloWorld");
```

Undefined ids

You can see that some statements must work with elements' ids. However, it's possible to ignore this requirement. Element's ids are used to keep references between program elements. They are necessary to perform any transformation. But if you want just to export the program tree into Java files, the ids will never be used.

Whenever the Java files are converted to an XML project, ids' structure is generated automatically. So it's possible to simplify our example by ignoring the ids' structure (and also some arguments). But we must not perform any transformation before we convert the tree into Java files and back to the XML project! Simplified version of our example:

```
 1   TreeInfo treeInfo = TreeUtils.createEmptyTree();
 2   JavaSourceProgram root = treeInfo.getRoot();
 3   ProgramElementFactory factory = treeInfo.getProgramElementFactory();

 4   Modifiers classModifiers =
     factory.createModifiers(factory.createModifier(ModifierType.PUBLIC));
 5   ClassDeclaration mainClass =
          factory.createClassDeclaration(classModifiers, "Main");
 6   JavaClassFile file = factory
          .createJavaClassFile("hello.world", mainClass);
 7   root.addJavaClassFile(file);

 8   Type argsType = factory.createType("String[]", TreeUtils.STRING_TYPE_ID);
 9   argsType.setDimensions("1");
10   argsType.setPrimitive("false");
11   FormalArguments arguments = factory.createFormalArguments
          (factory.createFormalArgument(argsType, "args", false));

12   Type returnType = factory.createType(TreeUtils.VOID_TYPE_NAME,
          TreeUtils.VOID_TYPE_ID);
13   Modifiers methodModifiers =
          factory.createModifiers(factory.createModifier(ModifierType.PUBLIC),
          factory.createModifier(ModifierType.STATIC));
14   Method mainMethod = factory.createMethod("main", returnType,
          methodModifiers, arguments);
15   Block block = factory.createBlock();
16   mainMethod.setBlock(block);
17   mainClass.addMethod(mainMethod);

18   int printlnDecl = treeInfo
          .getOuterDeclaration("java.io.PrintStream.println(java.lang.String)");
19   int systemDecl = treeInfo.getOuterDeclaration("java.lang.System");
20   int systemOutDecl = treeInfo.getOuterDeclaration("java.lang.System.out");

21   Type targetType = factory.createType("System", 0);
22   FieldRef targetField = factory.createFieldRef("out", 0);
23   targetField.setType(targetType);
24   Target callTarget = factory.createTarget(targetField);

25   Arguments printlnArgs = factory
          .createArguments(factory.createLiteralString("Hello World !!!"));
26   MethodCall printlnCall = factory.createMethodCall
     (callTarget, printlnArgs, "println", 0);
27   block.addStatement(printlnCall);

28   Marshal.marshall(root, "c:\\HelloWorld\\source.xml");
29   ToJava.transform(root, "c:\\HelloWorld");
```

C.2. Tutorial 2 – simple transformation

This tutorial shows how to implement a simple transformation using the XMLJT framework. Goal of this transformation is to print information about the change of watched variables into standard output. It means two statements must be added for each statement that changes value of the watched variable (printing value before the change and after the change). Let's have this program, we want to watch the variable *i*:

Original:

```
public class Main {
    public static void main(String[] args) {
        int i = 0;
        int j= 3;
        i = j + 9;
        while (i < 14)
            i++;
        j = i-1;
        i += 3;
    }
}
```

After transformation:

```
public class Main {
     public static  void main( String[] args) {
          int i = 0;
          int j = 3;
          System.out.println("Old value of variable i: " + i);
          i = j + 9;
          System.out.println("New value of variable i: " + i);
          while (i < 14) {
               System.out.println("Old value of variable i: " + i);
               i++;
               System.out.println("New value of variable i: " + i);
          }
          j = i - 1;
          System.out.println("Old value of variable i: " + i);
          i += 3;
          System.out.println("New value of variable i: " + i);
     }
}
```

The transformation has three parameters – source directory (containing Java source files), target directory (transformed source files will be generated there) and names of the variables to be watched (all local variables with given name will be watched). You can find the source code of this transformation in the *xmljt.transformations.tutorial.variablelogger.VariableLogger* class.

Import, transform, export

At first the Java source files must be converted into XML project (1-7). The file “project.xml” is created in the target directory (this directory must exist). All Java files in the source directory (including subdirectories) are parsed and added to XML file (3).

The user can get the root of the project tree by unmarshalling this XML file (8). This root is used to create an instance of the *TreeInfo* class that holds all information about

the structure of the tree (12). Instance of the *ProgramElementFactory* is stored to field to simplify the source code (13).

The visitor pattern is used to perform the transformation (14) – the *VariableLoggerVisitor* class will be described later. After performing any transformation *refreshTreeInfo()* method must be called (15). This method goes through the whole tree and refreshes information about relations between program elements.

When the source code has been transformed, the user will probably want to export it. It can be either exported to the XML project file (17) or the Java source files can be generated in the given directory (16).

```
   private TreeInfo treeInfo;
   private ProgramElementFactory factory;

   public void transform(String sourceDir, String distDir,String ... variables){
 1  String xmlFile = distDir + "\\project.xml";
 2  String error;
 3  error = JavaToXml.convert(sourceDir, xmlFile);
 4  if (error != null) {
 5       System.err.print("Error while parsing java files: " + error);
 6       return;
 7  }

 8  JavaSourceProgram root = Unmarshal.unmarshall(xmlFile);
 9  if (root == null) {
10       System.err.print("Error while reading xml file.");
11       return;
    }

12  treeInfo = new TreeInfo(root);
13  factory = treeInfo.getProgramElementFactory();
14  root.accept(new VariableLoggerVisitor(treeInfo, variables));

15  treeInfo.refreshTreeInfo();
16  ToJava.transform(root, distDir);
17  Marshal.marshall(root, xmlFile + ".new");
   }
```

Visitor implementation

The visitor pattern is used to go through the whole tree and to find the elements that should be transformed. The visitor class extends the *VisitAllElementsVisitor* class that automatically visits all elements of the tree. List of watched variables' names is passed to the constructor (21) and stored in the field (18).

To simplify this transformation let's say the variable can be changed only by an assignment or by a single unary expression (must not be nested within another expression).

Every time the visitor is visiting an assignment expression (27) it must ensure the target of the assignment is a local variable (28, 29). If the name of the local variable is contained in the set of variables to be watched (31), the method generating *println* statements is called (this method will be described later) (32).

The body of the *visitUnaryExpr* method is quite similar – if one of the watched local variables is being changed, the *generateLoggerStatemets* method is called (34-38).

```
   public static class VariableLoggerVisitor extends VisitAllElementsVisitor {

18  Set<String> varsToBeLogged;
19  private TreeInfo treeInfo;
20  private ProgramElementFactory factory;

21  public VariableLoggerVisitor(TreeInfo treeInfo, String ... variables) {
22       varsToBeLogged = new HashSet<String>(variables.length);
23       for (String v : variables)
24            varsToBeLogged.add(v);
25       this.treeInfo = treeInfo;
26       factory = treeInfo.getProgramElementFactory();
    }

    @Override
27  public void visitAssignmentExpr(AssignmentExpr x) {
28       if (!(x.getLvalue().getTarget() instanceof VarSet))
29            return;
30       VarSet varSet = (VarSet) x.getLvalue().getTarget();
31       if (varsToBeLogged.contains(varSet.getName()))
32            generateLoggerStatemets(x, varSet.getName(), varSet.getIdref());
    }

    @Override
33  public void visitUnaryExpr(UnaryExpr x) {
34       if (!(x.getExpression() instanceof VarRef))
35            return;
36       VarRef varRef = (VarRef) x.getExpression();
37       if (varsToBeLogged.contains(varRef.getName()))
38            generateLoggerStatemets(x, varRef.getName(), varRef.getIdref());
    }
   }
```

Create *println* statements

The method *generateLoggerStatemets* has got three parameters – statements that should be surrounded by the *println* statements, name of the variable to be printed out and the variable's id. The first step is to ensure the statement is the "real" statement (for example unary expression can be just the part of another expression – in that case we don't want to transform it) (40-41).

It’s necessary to know where the new statements will be added. The method *getStatementContainerForAdding* of the class *TreeInfo* returns the right statement container (42) – this method will create new container (block) if it’s necessary (if the parent of the statement is an instance of *Loop*, *DoLoop*, *TrueCase* or *FalseCase* class).

Now it’s easy to add two statements to surround the assignment (unary) expression. These statements are created by the *createPrintln* method (44-45).

```
   private void generateLoggerStatemets
               (Statement statement, String varName, int varId) {

39  Object parent = statement.getParent();
40  if (!(parent instanceof Loop || parent instanceof DoLoop ||
    parent instanceof TrueCase || parent instanceof FalseCase ||
    parent instanceof StatementContainer))
41             return;

42  StatementContainer stmtContainer =
    treeInfo.getStatementContainerForAdding(statement);
43  int stmtPos = stmtContainer.getStatements().indexOf(statement);
44  stmtContainer.insertStatement(stmtPos,
    createPrintln("Old value of variable " + varName +": ",varName,varId));
45  stmtContainer.insertStatement(stmtPos+2,
    createPrintln("New value of variable " + varName +": ",varName,varId));
   }
```

Creating the *println* method’s call is similar to the first tutorial (creating program elements). At first ids of the outer declarations must be loaded (46-48) and target of the method’s call must be created (49-52). The argument of the *println* must be a binary expression having the plus operator and two operands – string literal and reference to the watched variable (53, 54). Finally the newly created *MethodCall* statement is returned.

```
   private MethodCall createPrintln(String message,  String varName,
    int varId) {
46  int printlnDecl = treeInfo.getOuterDeclaration
    ("java.io.PrintStream.println(java.lang.String)");
47  int systemDecl = treeInfo.getOuterDeclaration("java.lang.System");
48  int systemOutDecl = treeInfo.getOuterDeclaration("java.lang.System.out");

49  Type targetType = factory.createType("System", systemDecl);
50  FieldRef targetField = factory.createFieldRef("out", systemOutDecl);
51  targetField.setType(targetType);
52  Target callTarget = factory.createTarget(targetField);

53  BinaryExpr arg = factory.createBinaryExpr("+",
        factory.createLiteralString(message),
        factory.createVarRef(varName, varId));
54  Arguments printlnArgs = factory.createArguments(arg);
55  return factory.createMethodCall(callTarget, printlnArgs,
        "println", printlnDecl);
   }
```

C.3. Tutorial 3 – adding a transformation to the XMLJT GUI

This tutorial shows how to add new transformations to the XMLJT GUI – it contains some built-in transformations but it's easily extensible.

The first step is to implement the *xmljtgui.transformations.ITransformation* interface (it can be found in *XMLJTgui.jar* file). Let's add the transformation from the previous tutorial to the GUI.

The user has to implement six methods. Methods *setTreeInfo* and *setFrame* are called by the XMLJT GUI every time the current tree info or frame is being changed. In our example this information is just stored to the fields (16, 17). The user also gets information about the current selection within the source code (4) – the start and the end position of the current selection and position of all elements (of the current file) within the source code. In the example this information is not used – you can use the source code of the *ITransformation* implementations in the *xmljtgui.transformations* package to learn how to use it.

The *getName* method returns the name that will be displayed in the menus of the graphic user interface (3). This menu item will be grayed if the *canExecute* method returns false. In the example the transformation is enabled every time there is some project opened (5).

The most important method to be implemented is named *execute*. It's invoked when the user clicks on the corresponding menu item. The error message is displayed

using the *DialogOutput* class if there is no opened project (7 -10). Standard Swing input dialog is used to get the name of the variable to be logged (11-13). Now the transformation can be finally performed using the visitor from the previous tutorial (14). The method *execute* returns true if the transformation has been performed successfully (15).

```
  public class LogVariables implements ITransformation {

 1  private TreeInfo treeInfo;
 2  private JFrame frame;

 3  public String getName() {
        return "Log Variables";
    }

 4  public void setSelection(int start, int end, List<ElementPosition>
        elementsPositions) {
        // do nothing
    }

 5  public boolean canExecute() {
       return treeInfo != null;
    }

 6  public boolean execute() {
 7      DialogOutput output = new DialogOutput(frame);

 8      if (!canExecute()) {
 9          output.error("No source code");
10          return false;
        }

11      String name = JOptionPane.showInputDialog("Name of variable to
                        be watched:");
12      if (name == null)
13          return false;

14      treeInfo.getRoot().accept(new VariableLoggerVisitor(treeInfo,
               new String[]{name}));

15      return true;
    }

16  public void setTreeInfo(TreeInfo treeInfo) {
        this.treeInfo = treeInfo;
    }

17  public void setFrame(JFrame frame) {
        this.frame = frame;
    }
  }
```

This implementation of the *ITransformation* interface must be build into a jar file. Let's name this file "varlogger.jar". All extensions must be stored in the directory

"transformations" that must be in the same directory as the executable *XMLJTgui.jar* file. If there is no such directory it must be created. The jar file containing the extension (*varlogger.jar*) must be copied to this directory.

Information about extensions is stored in the text file named "transformations.cfg". If the user wants to add the first extension he has to create this file (using any text editor). This file contains full names (separated by a new line) of classes that implement the *ITransformation* interface and should be added to the user interface. Configuration file (*transformations.cfg*) of the example will have this content:

```
xmljtgui.transformations.LogVariables
```

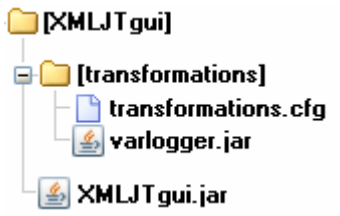


Figure 8: XMLJT GUI extensions hierarchy

Now it's possible to run our transformation from the GUI (using pop-up menu or main menu).