Vrije Universiteit Brussel – Belgium

Faculty of Sciences
In Collaboration with Ecole des Mines de Nantes – France
and
University of Twente – The Netherlands
2003

A Representation of Java Programs as Partial Graph Morphisms

A Thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
(Thesis research conducted in the EMOOSE exchange)

By: Angela Consuelo Lozano Rodríguez

Promoter: Prof. Theo D’Hondt (Vrije Universiteit Brussel)
Co-Promoter: Prof. Arend Rensink (Universiteit Twente)

Table of Contents

1. Introduction 5

1.1. GOALS 5
1.2. CONCEPTS 6
1.2.1. Graphs 7
1.2.2. Transformation & production rules 7
1.2.3. Double push out 9
1.2.4. Single push out 9
1.2.5. Negative application conditions 10
1.2.6. Graph grammar 10
1.3. JAVA SOURCE STRUCTURE 10
1.3.1. Declarations 11
1.3.1.1. Abstract modifier 11
1.3.1.2. Final modifier 11
1.3.1.3. Native modifier 11
1.3.1.4. Package modifier (none/default) 11
1.3.1.5. Private modifier 12
1.3.1.6. Protected modifier 12
1.3.1.7. Public modifier 12
1.3.1.8. Strictfp modifier 12
1.3.1.9. Static modifier 12
1.3.1.10. Synchronized modifier 12
1.3.1.11. Transient modifier 13
1.3.1.12. Volatile modifier 13
1.3.2. Types 13
1.3.3. Fields 14
1.3.4. Methods 14
1.3.4.1. Parameters 14
1.3.4.2. Throws clause 15
1.3.4.3. Method body 15
1.3.4.3.1. Variables 15
1.3.4.3.2. Expressions 15
1.3.4.3.3. Literals 16
1.3.4.3.4. Operators 16
1.3.4.3.5. References 18
1.3.4.3.6. Statements 18
1.4. CONTEXT 19
1.4.1. Groove 20
1.4.2. Recoder 21

2. Principles of translation 22

2.1. GRAPH NOTATION 22
2.1.1. Nodes definition 22
2.1.2. Edges definition 25
2.1.3. Object oriented concepts as graphs 32
2.2. CLASS GRAPHS 33
2.3. OBJECT GRAPHS 35
2.4. PATTERN RULES 37
2.4.1. Literal creation 37
2.4.2. Variable declaration 38
2.4.3. Operations 40
2.4.4. Assignment 42

2.4.5. Methods 43
2.4.5.1. Method environment creation 45
2.4.5.2. Method call 46
2.4.5.3. Method return 47
2.4.6. If 49
2.4.7. While 54

3. Tool description 58

3.1. USER GUIDE 58
3.1.1. System requirements 58
3.1.2. Installation instructions 58
3.1.3. Instructions of use 59
3.1.4. Output directory structure 62
3.1.5. Reporting bugs 63
3.2. PROGRAMMER GUIDE 65
3.2.1. Requirements 65
3.2.1.1. Project risks 65
3.2.2. Background 66
3.2.2.1. Groove 66
3.2.2.2. Recoder 68
3.2.3. Translator architecture 70
3.2.4. Subsystem design 71
3.2.4.1. Model package 71
3.2.4.2. Reference package 72
3.2.4.2.1. Field 73
3.2.4.2.2. Class 73
3.2.4.2.3. Method 74
3.2.4.2.4. Super constructor 74
3.2.4.2.5. Super 74
3.2.4.2.6. This constructor 75
3.2.4.2.7. This 75
3.2.4.2.8. Type 75
3.2.4.2.9. Variable 76
3.2.4.3. Rule patterns package 76
3.2.4.4. Gui package 77
3.2.5. Getting involved in the project 78
3.2.5.1. How to join the project 78
3.2.5.1.1. Last cvs version 78
3.2.5.1.2. Directory structure 79
3.2.5.2. How to extend the project 80
3.2.5.2.1. Adding new production rules 80
3.2.5.2.2. Changing parser 80

4. Processing algorithms 81

4.1. PARSING AND ABSTRACT SYNTAX TREE CONSTRUCTION 81
4.2. PREPROCESSING 82
4.3. PRODUCTION RULES GENERATION 83

5. Future work proposals 85

5.1. MISSING OR INCOMPLETE FEATURES 85
5.2. IMPROVEMENTS 86
5.3. PRIORITIES 87

6. Conclusions 89

7. References 91

How to read this document

This document pretends to summarize the work done as graduation project that defines how to map an object oriented language as Java into graphs and how the tool that performs this mapping was done.

These graphs, in particular, have non labeled nodes and labeled edges. However, the edges' label doesn't represent any identity, i.e. they can only be differentiated by their components: source node, label and target node. In this sense we can call them flat graphs because their components do not have identity by themselves.

The text is organized as follows: the first chapter summarizes the basic points in which the work was based on, like its objectives, background theory and tools used. Chapter 2 explains the proposed translation from Java source code to *flat graphs*. Chapter 3 describes the tool developed to perform the translation. Chapter 4 explains in detail how the software works and the necessary knowledge to extend it. The proposals for future work, features to implement or complete, as well as the possible improvements and their priorities are located in the chapter 5. The conclusion indicates which goals were reached and which was the contribution of this work.

1. Introduction

This thesis is intended to model Java source code as graph transformations within a larger project called GROOVE (see section 1.4.1) oriented to develop a fully automated verification tool for object oriented programs.
In a wider perspective the graph transformations that generate the software developed give the possibility to generate all possible transitions from an initial state represented as a graph, this is called a *transition system*. From this set of possible states it is feasible to extract canonical patterns that represent a set of patterns. This generalizations are called *temporal logic graphs* and they represent the behavioral semantic of a set of transformations. In this way, they enable automatic verification of a program [3].
In the project, there is another master thesis being developed whose goal is to translate Java byte code into graph transformations. Our work crossed in order to define some parameters on the produced graph transformations in order to make them compatible. We defined an interface / standard of graph transformations that will derive the same kind of state graphs.

The tool product of this thesis (called Translator) was developed as an extension of GROOVE. GROOVE (GRaphs for Object Oriented VErification) (see section 1.4.1) is a set of tools product of a straightforward translation from the graph rewriting theory. GROOVE software (version 0.0.1) is composed by an **editor** that allows the creation of state graphs and production rule or transformation graphs, by a **viewer** that shows with a special format a production rule and by a **simulator** that, given an initial state graph and production rule graphs, can show the effects of a production rule applied to a state graph and also produce all possible states starting with the initial state and applying the production rules to this initial state or to the intermediate generated states.

1.1. Goals

The main interest of this work is to represent object oriented programs in graphs whose elements do not have own identity. In particular, Java source code. For this reason one of the higher priority tasks was to study the Java specification in

order to get a view of the possibilities that this language offers, and the scope that could be covered in the present work.

Once the translation between the Java sources entities is defined it is also necessary to translate each execution point into a graph transformation. The objective was to identify all entities in the code that could generate changes in the object and represent this change as a graph transformation. This change representation implies the introduction of other entities that allow some restrictions not offered by source code entities themselves; but by the way they are interpreted such as the order of instructions. Note that the graph transformations are not just simple mapping from the source code to graphs because there are execution features implicit in the source code like class loading or instruction sequence; the mapping from source code to graphs should produce the same effect as the code execution would do in an object. It is necessary to introduce some execution information based only on static information provided by the source code.

In the end the produced transformation graphs must simulate the code execution, registering in the state graphs all changes in the objects involved in a program.

Summarizing, this project intends to:

- Construct a model that translates Java source code into flat graphs.
- Enrich the model to include implicit Java source features like class loading, garbage collection, execution order, etc.
- Identify the transformation points in a Java source code.
- Build canonical patterns for object oriented code transformations.
- Develop a proof of concept tool that extends GROOVE and generates graph transformation rules from Java source code

1.2. Concepts

This section comprises in an informal and condensed way all necessary terms used along the document. It explains the main theoretical concepts in which this work is based on.

1.2.1. Graphs

Graphs are mathematical representations that can be formally described, interpreted and transformed; furthermore they have intuitive visual representation. The graphs are composed by 2 kinds of elements: the nodes and the edges. The edges are elements that connect nodes. See [3].

The graphs that concern this work are not node labeled but edge labeled. Nevertheless neither nodes nor edges have identity of their own. Edges are distinguished by its components: source node, label(s) and target node. Two edges cannot have the same label and nodes.

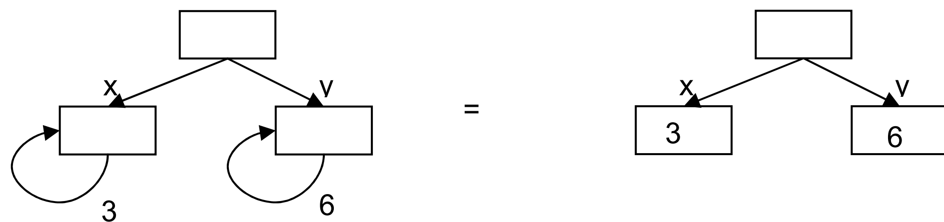


Figure 1.2.1.1. Two representations of a graph with 3 nodes and 4 edges.

The nodes are graphically depicted as boxes, the edges as directed arrows that indicate the starting and ending edge except in self referenced edges i.e. edges whose source and target node are the same. They are represented with the label of the self referenced edge inside the source-target node.

1.2.2. Transformation & Production rules

A transformation is a way to represent the change from an initial graph to a final graph.

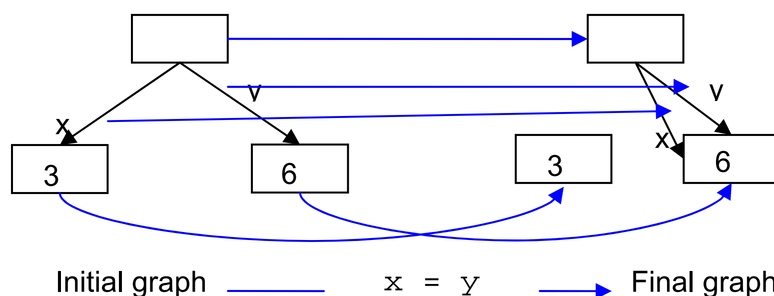


Figure 1.2.2.1. A transformation instance of x = y

The generalization of a transformation is a pattern that can be applied to different initial graphs and when they are applied can produce different final graphs. These patterns are called production rules. The productions rules indicate the nodes and edges created or removed from the initial graph to the final graph. The figure 1.2.2.2 shows a production rule in which the edge identified with the label x is deleted in the final graph, and the edge identified with an x label in the final graph does not have a pre image in the initial graph, this gain or loss of elements in the transformation is represented with question marks.

A rule only can be applied if it matches in the initial graph. A rule can have multiple matchings in a graph and its result depends in which of the matchings the pattern is applied. Any rule application generates a new graph.

The rule can also be defined as a partial injective function whose domain is the initial graph and its co domain is the final graph.

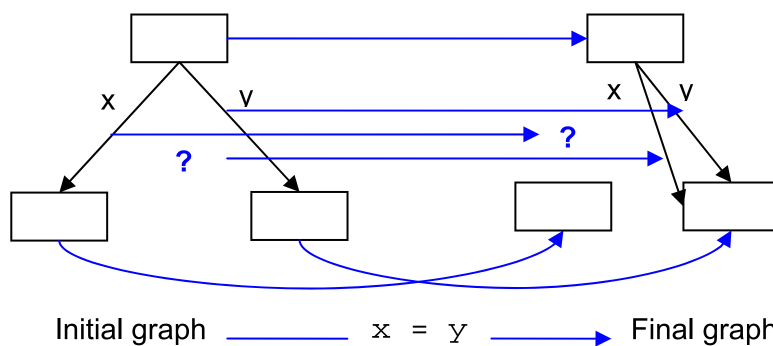


Figure 1.2.2.2. A transformation pattern of x = y

There are 4 element roles in a graph transformation. The *reader* element that must be in the initial and final graph, the *eraser* element that is present in the initial graph but not in the final graph, the *creator* that is present only in the final graph and the *embargo* element that cannot be present in the initial graph.

If different elements on the rule correspond to the same element on the graph, the element will be transformed with the most powerful transformation that is applied to it. For instance, let's suppose there is a rule that requires all self referenced nodes an deletes all nodes target of an edge labeled x. If the source graph has a self referenced node with an x, this element will match with a reader role and also with an eraser role; then the rule application will delete this node.

1.2.3. Double push out

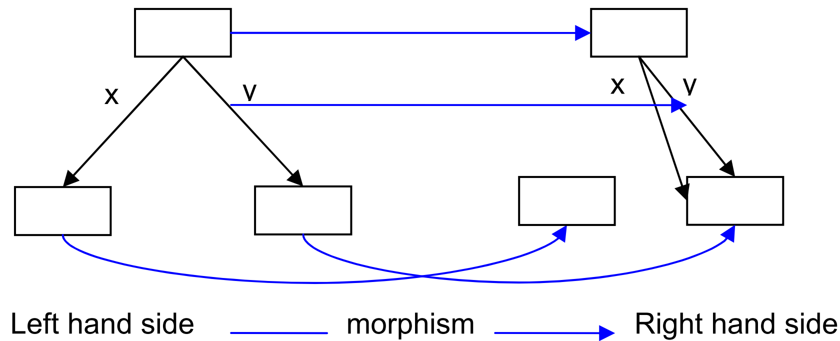
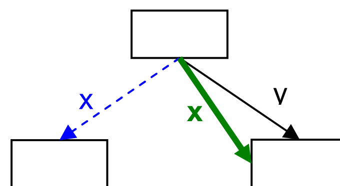


Figure 1.2.3.1. Double push out graphical representation. Transformation rule for x = y

The classical representation of a production rule is called *double push out*. It consists of the initial graph called also the left hand side graph (LHS), the final graph or right hand side graph (RHS) and the morphism between them. This morphism is a partially injective function, which establishes which elements of the LHS have which image in the RHS (if they have). See [1].

1.2.4. Single push out

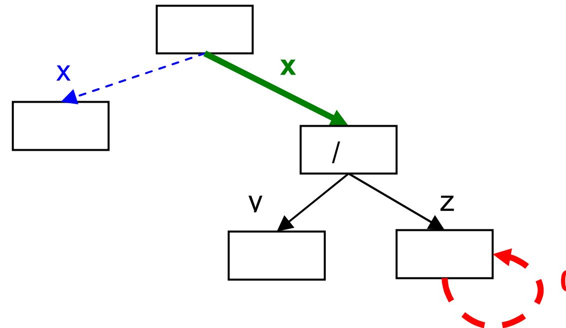


Figrure 1.2.4.1. Single push out in GROOVE's output format. Transformation rule for x = y

The double push out representation can be summarized indicating the necessary nodes to have a match, which of them are deleted in the RHS graph and which are created, this is, elements that do not have an image in the LHS. This representation is just the disjoint union between LHS and RHS. This equivalent function constructs the set of elements of LHS that are not in the co domain of the morphism (nodes or edges deleted) and the set of elements of RHS that are not in the in the domain of the morphism (nodes or edges created). It is called the

single push out approach. Translator represents production rules with the single push out approach. See [1] and [4].

1.2.5. Negative Application Conditions



Figrure 1.2.4.1. Single push out enriched with a Negative Application Condition in GROOVE's output format. Transformation rule x = y / z

The single push out graphs are enriched with a kind of Negative Application Conditions (NAC) that forbid relations in the initial graph, in other words, the rule is not applicable if there is a match of the NAC in the initial graph. The role played by the elements that are part of the NAC are called embargo. See [4].

Note that due to a NAC defines relationships, the elements that compose it are edges. Then the NAC elements are also called embargo edges.

1.2.6. Graph grammar

The duple formed by an initial state graph and a set of production rules is called graph grammar.

A graph transition system if a triple formed by an initial state graph, a set of transition rules and a set of final states. See [4].

A graph grammar produces a transition system if the transitions correspond to all possible rule applications to all reachable states.

1.3. Java source structure

This section is dedicated to present an overview of the main structural nodes that compose a Java program. The immediate interest is to condense all necessary terms that will be mention now and on from a source code point of view. This section can be omitted if you are familiar with the Java Language Specification. See [2].

1.3.1. Declarations

A source code is a set of declaration that establishes new named program elements. In general, a source file contains one or more class declarations, with their corresponding identifying information and its members: accessibility, name, super class, implemented interfaces, fields, and method declarations. Which at the same time have their own identifying and member information like accessibility, name, exceptions thrown, parameters, local variables, etc. What is more, each object mentioned in a source file must have its own declaration indicating at least its name and type, in some cases its initial value. With the exception of the literals, a literal is a source code representation of a value of primitive type.

The most important concept present in a declaration is the modifier. The modifiers are reserved keywords that restrict the use of a program element (***abstract****;* ***final****;* ***native****;* ***private****;* ***protected****;* ***public****;* ***static****;* ***strictfp****;* ***synchronized****;* ***transient****;* ***volatile***). The modifiers vary depending on the program element declared. For instance a class declaration only can have abstract, final and strictfp modifiers, a field declaration can be static, final, transient and volatile, and a method declaration can be abstract, static, final, native, strictfp, synchronized.

1.3.1.1. Abstract modifier

In a class declaration it means that the class cannot be instantiated and it may contain unimplemented methods. All interfaces are abstract. In a method declaration, means that it does not have a body and the enclosing class is abstract.

1.3.1.2. Final modifier

The final modifier means that the program element cannot be changed; this implies that a final class cannot be sub-classed, a final method cannot be overridden and dynamically looked up, a final variable or final field cannot change its value. The static final fields are compile-time constants.

1.3.1.3. Native modifier

It can only appear in a method declaration and means that it is platform-dependent. This method does not have body, only signature.

1.3.1.4. Package modifier (none/default)

It means that the program element is accessible only inside its package. It can appear in class, interface and method declarations.

1.3.1.5. Private modifier

It means that the program element is a class member accessible only in the class which defines it.

1.3.1.6. Protected modifier

It means that the program element is a class member accessible only within the subclasses and package of the class which defines it.

1.3.1.7. Public modifier

It means that the program element is accessible anywhere. It can appear in class, interface and member declarations (in these cases the member is accessible anywhere its class is).

1.3.1.8. Strictfp modifier

It means that all floating-point computation done is strictly conforms to the IEEE 754 standard. All values including intermediate results must be expressed as IEEE float or double values. It can appear in class and method declarations. A class strictfp implicitly has all its methods strictfp.

1.3.1.9. Static modifier

It means that a class has members that can be accessed without a class instance, because they do not depend of the instance's state. It can appear on a class, method, field or initializer declarations. A static method is also called a class method, and can be invoked through the class name.
A static or class field (exists only one instance of this field for all class instances), can be invoked through the class name, regardless of class instances created. A static modifier inside an initializer indicates that it is executed when the class is loaded, rather than when an instance is created.

1.3.1.10. Synchronized modifier

It can only appear in method declarations. For a static method, a lock for the class is acquired before executing the method. For a non-static method, a lock for the specific object instance is acquired.

1.3.1.11. Transient modifier

It can only appear in field declarations. It means that the field does not belong to the persistent state of the object and they will not be serialized with the object.

1.3.1.12. Volatile modifier

It can only appear in field declarations. It means that the field is accessible by unsynchronized threads, i.e. each thread can have a working copy of the field.

1.3.2. Types[1]

Java is a strongly typed language what implies that every variable has a known type at compile time. The type determines the structure and operations of each object.

Java types are classified into two kinds: the primitive and the reference types.

The primitive types are predefined and have reserved keywords. They are subdivided in numeric and ***boolean*** types. The numeric types are also subdivided in integral types and floating point types. The integral types are: ***byte***, ***short***, ***int***, ***long*** and ***char***. The floating point types are: ***double*** and ***float***.

The reference types are the classes and interfaces. They are defined in compilation units. They can have a modifier, a super type, variables and methods. The referenced types can be instantiated or not. The types that cannot be instantiated are the interfaces and the abstract classes.

The interfaces can be public, protected or private, and its super type can only be an interface. Its methods cannot have an implementation and its variables must be static final. The abstract classes can have abstract and non abstract methods.

The types that can be instantiated are divided into final and non final types. The final types cannot have subclasses and all its methods must be final. The non final types only can contain non abstract methods.

Another kind of type is the array type. The array types do not have super types. They define a set of fixed length that contains components of the same type (called component type). Array types can have as component type other array types.

All non primitive types can have only one super type. This inheritance relation creates a hierarchical type structure whose root is the class Java.lang.Object that does not have super type.

[1] All types that are not interfaces are called classes.

Each class has one or more constructor methods and can have zero or more instance variables called fields.
In order to recognize uniquely a type the fully qualified name that is composed by the package name and the type name is used. A type package is defined by the package declared in its container compilation unit[2]. The package is a hierarchical organization of the type names, a package can be nested in other packages. The package name is defined by the containing package name followed by a dot followed by the package name.

1.3.3. Fields

The fields are variables that belong to a class (i.e. the field is static then there is just one object for all class instances) or to an instance (i.e. the field is not static then it exists one object for each class instances). As all variables they have a name and type.
All fields inside a class must have a different name and can have a different accessibility.

1.3.4. Methods

The methods are composed by their modifier, return type, signature, throw clause and their body. A type cannot have two methods with the same signature. There are two kinds of methods, those that do not have a return type called constructors and the regular methods that have return type. The constructors' names are the same of their containing class. Each class has at least one constructor method, if it is not defined explicitly the language will create a default one without parameters and whose accessibility will be the same as its container class.
The method signature is composed by the method name and its parameters.

1.3.4.1. Parameters

The parameters are method variables (i.e. they have a type and a name). A method can have zero or more parameters but they cannot have the same name. Method parameters name argument values passed to a method. For every parameter declared in a method declaration, a new parameter variable is created each time that method is invoked. The new variable is initialized with the corresponding argument value from the method invocation. The method

2 A compilation unit is an abstract program element (equivalent to a Java file) in which can be defined several types.

parameter effectively ceases to exist when the execution of the body of the method is complete.

1.3.4.2. Throws clause

The throws clause indicates the possible exceptions raised by the method execution. All types thrown must be subclasses of java.lang.Throwable.

1.3.4.3. Method body

The method body is composed by local variable declarations, expressions, literals, operators, references and statements.

1.3.4.3.1. Variables

The local variables can only be accessible inside the method scope.
Local variables are declared by local variable declaration statements. Whenever the flow of control enters a block or for statement, a new variable is created for each local variable declared in a local variable declaration statement immediately contained within that block or for statement. A local variable declaration statement may contain an expression which initializes the variable. The local variable with an initializing expression is not initialized, however, until the local variable declaration statement that declares it is executed. (The rules of definite assignment prevent the value of a local variable from being used before it has been initialized or otherwise assigned a value.) The local variable effectively ceases to exist when the execution of the block or for statement is complete.

1.3.4.3.2. Expressions

Every expression written in the Java programming language has a type that can be deduced from the structure of the expression and the types of the literals, variables, and methods mentioned in the expression. The language performs an implicit conversion from the type of the expression to a type acceptable for its surrounding context.
Then the five conversion contexts are:

- Assignment conversion converts the type of an expression to the type of a specified variable. The conversions permitted for assignment are limited in such a way that assignment conversion never causes an exception.
- Method invocation conversion is applied to each argument in a method or constructor invocation and, except in one case, performs the same

conversions that assignment conversion does. Method invocation conversion never causes an exception.

- Casting conversion converts the type of an expression to a type explicitly specified by a cast operator. It is more inclusive than assignment or method invocation conversion, allowing any specific conversion other than a string conversion, but certain casts to a reference type may cause an exception at run time.
- String conversion allows any type to be converted to type String.
- Numeric promotion brings the operands of a numeric operator to a common type so that an operation can be performed.

1.3.4.3.3. Literals

A literal is the source code representation of a value of a primitive type, the String type, or the null type.

An integer literal may be expressed in decimal (base 10), hexadecimal (base 16), or octal (base 8).

A floating-point literal has the following parts: a whole-number part, a decimal point (.), a fractional part, an exponent, and a type suffix.
At least one digit, in either the whole number or the fraction part, and a decimal point, an exponent, or a float type suffix are required. All other parts are optional.

The boolean type has two values, represented by the literals true and false, formed from ASCII letters.

A character literal is expressed as a character or an escape sequence, enclosed in ASCII single quotes (‘).

A string literal consists of zero or more characters enclosed in double quotes. Each character may be represented by an escape sequence.

The null type has one value, the null reference, represented by the literal null, which is formed from ASCII characters.

1.3.4.3.4. Operators

The Java operators are :

- The comparison operators < , <= , >, and >= .
- The equality operators == and !=
- The logical-complement operator !
- The logical operators & , ^ , and |
- The conditional-and and conditional-or operators && and ||
- The conditional operator ? :
- The numerical operators:
    - The unary plus and minus operators + and -
    - The multiplicative operators * , / , and %
    - The additive operators + and -
    - The increment operator ++ , both prefix and postfix
    - The decrement operator -- , both prefix and postfix
    - The integer bitwise operators & , | , and ^
- The signed and unsigned shift operators << , >> , and >>>
- The bitwise complement operator ~
- The cast operator, which can convert from an integral value to a value of any specified numeric type
- The operators on references to objects:
    - Field access, using either a qualified name or a field access expression
    - Method invocation
    - The cast operator
    - The instanceof operator
- The string concatenation operator +, which, when given a String operand and:
    - an integral operand, will convert the integral operand to a String representing its value in decimal form, and then produce a newly created String that is the concatenation of the two strings
    - a floating-point operand, will convert the floating-point operand to a String representing its value in decimal form (without information loss), and then produce a newly created String by concatenating the two strings
    - a boolean operand, will convert the boolean operand to a String (either "true" or "false"), and then produce a newly created String that is the concatenation of the two strings

- a reference, will convert the reference to a String by invoking the toString method of the referenced object (using "null" if either the reference or the result of toString is a null reference), and then will produce a newly created String that is the concatenation of the two strings

1.3.4.3.5. References

The references are elements representing implicit or explicit (named) references to other program elements.

1.3.4.3.6. Statements

The statements are control flow structures. The Java statements are: *break*, *continue*, *do*, *empty statement*, *expression statement*, *for*, *if*, *labeled statement*, *return*, *switch*, *synchronized*, *throw*, *try*, *while*.

The *break* statement transfers control out of an enclosing statement.

A *continue* statement may occur only in a while, do, or for statement; statements of these three kinds are called iteration statements. Control passes to the loop-continuation point of an iteration statement.

The *do* statement executes a Statement and an Expression repeatedly until the value of the Expression is false.

An *empty* statement does nothing.

An *expression* statement (Assignment; PreIncrementExpression; PreDecrementExpression; PostIncrementExpression; PostDecrementExpression; MethodInvocation; ClassInstanceCreationExpression;) is executed by evaluating the expression; if the expression has a value, the value is discarded. Execution of the expression statement completes normally if and only if evaluation of the expression completes normally.

The *for* statement executes some initialization code, then executes an Expression, a Statement, and some update code repeatedly until the value of the Expression is false.

The *if* statement allows conditional execution of a statement or a conditional choice of two statements, executing one or the other but not both.

A *labeled* statement is executed by executing the immediately contained *Statement*. If the statement is labeled by an *Identifier* and the contained *Statement* completes abruptly because of a break with the same Identifier, then the labeled statement completes normally. In all other cases of abrupt completion of the *Statement*, the labeled statement completes abruptly for the same reason.

A *return* statement returns control to the invoker of a method or constructor

The *switch* statement transfers control to one of several statements depending on the value of an expression.

A *synchronized* statement acquires a mutual-exclusion lock on behalf of the executing thread, executes a block, then releases the lock. While the executing thread owns the lock, no other thread may acquire the lock.

The *throw* statement causes an exception to be thrown. The result is an immediate transfer of control that may exit multiple statements and multiple constructor, instance initializer, static initializer and field initializer evaluations, and method invocations until a try statement is found that catches the thrown value. If no such try statement is found, then execution of the thread that executed the throw is terminated after invocation of the uncaughtException method for the thread group to which the thread belongs.

A *try* statement executes a block. If a value is thrown and the try statement has one or more catch clauses that can catch it, then control will be transferred to the first such catch clause. If the try statement has a finally clause, then another block of code is executed, no matter whether the try block completes normally or abruptly, and no matter whether a catch clause is first given control.

The *while* statement executes an *Expression* and a *Statement* repeatedly until the value of the *Expression* is false.

1.4. Context

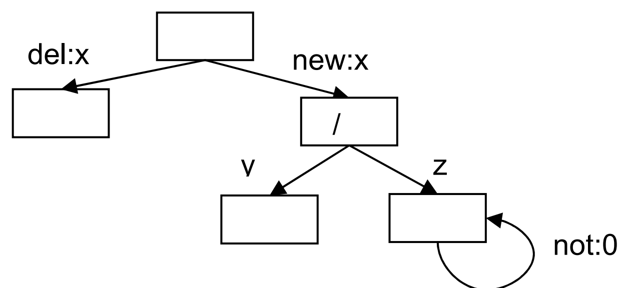
This section gives some detail about the tools used in order to build Translator.

Translator is an extension of GROOVE. GROOVE is a software project whose purpose is to verify the semantic of object oriented programs.
The parsing and Abstract Syntax Tree generation was delegated to another tool called Recoder.

1.4.1. GROOVE

GROOVE (GRaphs for Object Oriented VErification) is a tool for representing the semantics of object-oriented programs using graphs in order to perform automatic verification and properties validation. GROOVE is developed with Java jdk 1.4, in its actual version 0.0.3 it includes an editor for creating production rules, a viewer for visualizing the production rules and a simulator for applying the graph transformations of a set of production rules. The system was developed by Arend Rensink at the software engineering group at University of Twente. See [4].

GROOVE takes advantage of the visual representation of graphs and adds some semantics to it. In particular, it provides a different visualization for each kind of the roles that an element can be playing in a production rule. The reader or required elements are depicted with solid thin black arrows and boxes. The eraser elements are represented with dashed thin blue lines and boxes. The embargo elements are showed with dashed fat red arrows. And the creator elements are depicted by solid fat green arrows and nodes.
This representation is enabled just once the production rule is saved. Meanwhile, in the editor there are some special prefixes that specify the role of each element. The reader or required elements prefix is “use:”, the eraser elements prefix is “del:”, the embargo elements prefix is “not:” and the creator elements prefix is “new:”. If the element does not have any prefix the editor will assume it as a reader node as default.



Figrure 1.4.1.1. Single push out enriched with a Negative Application Condition in GROOVE’s input format. Transformation rule x = y / z

GROOVE's output format is XML, the XML text files categorize the rule elements in terms of their role prefix.

1.4.2. Recoder

Recoder is a framework for Java meta programming, it provides elements to analyze and transform Java code. The core system was developed by Andreas Ludwig as part of his PhD thesis, with help and support from Uwe Aßmann and co-author Rainer Neumann at the software engineering and compiler construction group of Prof. Dr. Goos at the University of Karlsruhe. See [5].

The developed tool benefits from the Recoder parsing facilities. Recoder assumes that the input is a syntactically correct Java source file (i.e. it can be compiled without errors) and creates a model unambiguously based on a partial semantic analysis. The model is retrieved in an Abstract Syntax Tree.

These are some important characteristics that make Recoder a special parser:

Semantic entities (Type, Variable) are distinguished from their definitions (TypeDeclaration, VariableDeclaration) and uses (TypeReference, VariableReference).
The language specification distinguishes between type (of variables and expressions only) and class (of objects during execution). Recoder is not so strict.

Addition of parent references to any syntactical elements. It allows easy access to arbitrary program elements as argument for a program transformation; avoids the need to traverse the trees to find the context of a program element. This direct access to syntax elements requires quick access to parents, which should be type safe.
Note: recoder.Java.expression and recoder.Java.statement contain pure expressions and statements, respectively, but they do not contain all of them; many references are valid expressions, some of them are hybrids (ExpressionStatements).
Note: Not all references are expressions, e.g. PackageReference, TypeReference.

2. Principles of translation

This chapter describes the model created to represent the source code in graph terms.
In first place, it illustrates the notation used to construct these graphs. In second place, it explains the transformation concepts.
The transformation process is accomplished in a two level transformation: a preprocessing transformation that maps the class defined in the Java source code into the proposed Java graph model, and the generation of the transformation rules, obtained by applying the Java source code instructions to an object graph (i.e. an instance of the class graph produced in the first transformation stage).

This chapter is divided into 4 sections. The first section explains the graph elements designed to map Java source code into graphs, i.e. the model that maps Java model into flat graphs.
Sections 2.2 and 2.3 explain how the model explained in section 2.1 is used to perform the translation from the Java source to a graph. Simply put, the second section shows a class graph example, while the third section shows an object graph example.
Section 2.4 describe in detail the mapping of Java instructions to production rules and how they are instantiated to obtain the production rules that represent certain Java program.

2.1. Graph notation

The graph representation of the Java entities is straightforward from the source code. The software units (classes, objects, methods, packages, statements, etc) are represented as nodes identified by its name and its type (or the entity that represent the name or type). While the software relations (inheritance, aggregation, ownership relations, etc.) are represented as edges.

2.1.1. Nodes definition

In this section, all node types defined to model Java code static information are listed .
The model created for characterizing typed languages is achieved by representing in the same level objects and classes.

A class represents the common properties of all its instances. In this model a class is represented by a node with a self reference edge labeled with the full qualified name of the class. As shown in the figure 2.1.1.1

Figure 2.1.1.1. Class node example

The method node represents each one of the methods defined in a particular class. In order to distinguish among them they are labeled not just by their name but also by the type of their parameters (in the order they are defined). In a Java way like, that is to say, the types are replaced by its Java VM Type Signatures[3]. The figure 2.1.1.2 shows a method node for the method of the form:

```
String toString(){….}
```

Figure 2.1.1.2. Method node example

The object node corresponds to an instance of a class; it encloses its identifying and runtime information. As they are not self labeled with a particular name they can be referenced with labeled edges; this means that the objects are managed in a pointer like way. The figure 2.1.1.3 depicts an object node called size.

Figure 2.1.1.3. Object node example

[3] The general form of a Java method signature is: "(argument-types)return-type".
The encoding for the Java VM Type Signatures is: Z for boolean, B for byte, C for char, S for short, I for int, J for long, F for float,D for double, L for fully-qualified-class and [for an array type
For instance, the signature (I)V, for example, denotes a Java method that takes one argument of type int and has a return type void.

The method instance nodes represent each one of the calls of a specific method. Like the object nodes, the method instance nodes do not have any identifier. They are necessary because each object can call the methods defined by its class but each method call should have specific state depending on the state of the object that calls it, the value of the parameters passed, and the exact execution point in which they are.

This exact point in the execution of each method is modeled with the instruction order node, an example of it can be seen in figure 2.1.1.4. The goal is to maintain the sequence of instructions determined by the source code.

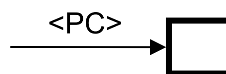


Figure 2.1.1.4. Instruction order node example

The Java Virtual Machine (see figure 2.1.1.5) node was created in order to simulate the class loading and garbage collection. When a class is loaded is created a edge from the JVM node to the class node, in this way, each class is loaded just once and is easy to recreate the following reference algorithm do perform garbage collection (the nodes not referenced are deleted, the garbage collection starts to mark the referenced objects starting with the JVM references)

Figure 2.1.1.5. Instruction order node example

The nodes definition table (2.1.1.1) recapitulates the type of nodes defined to represent Java source code. The first column enumerates the represented software units. The second says if the node has an identifier (self reference with a particular label). The identifier can have two values: none (i.e. it does not have self referenced label) or the information that represents the label of this kind of nodes. The third column has an abbreviation for each kind of nodes in order to obtain a canonical textual representation of the defined graphs.

Type	Identifier	Abbreviation[4]
Class	Fully qualified name of the class	**C, I (interface)**
Method	Java signature of the method	**M**
Object	None	**O**
Method instance	None	**MI**
Instruction order	Index of the instruction	**IO**
Operation[5]	Operation identifier	**OP**
Statement[6]	Statement name	**S**
Java Virtual Machine	JVM	**JVM**

Table 2.1.1.1. Nodes definition

2.1.2. Edges definition

In this section, all edge types defined to model Java code information are listed. These edges complete the model by describing relations between object oriented entities and adding dynamic / execution information.

The super edge connects a class node with its corresponding super class, and also an object node with its corresponding object node (i.e. an instance of its super class that will represent its super node, thus, there is a clear separation between the object level and the meta object level in the graph representation). This differentiation is exposed in the figures 2.1.1.2 and 2.1.1.4. Note that this distinction is very important in order to be capable of distinguishing among objects. For example, the fields are edges from the object container to the object contained labeled with the name of the field. As the fields belong to the object and not to the class each object must have a chain of super objects to offer a clear mapping from the meta model to the object space. As shows the figure 2.1.1.2.

[4] This abbreviation is just a textual simplified representation of each one of the types of nodes, in order to explain in a compact way its possible relationships in the edges' table.
[5] The operation nodes represent the primitive transformations like addition, times, division, modulo, shift, logical and, etc.
[6] A statement node represents a control instruction of the language like if, for, while, etc.
The operation node and the statement node are equivalent representations of the method instance node. They encapsulate primary transformations offered by the language in a defined semantic that allow its simulation.

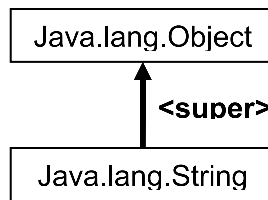


Figure 2.1.1.2. Super class edge example

The instanceOf edge allows the connection between the meta-model to the model by connecting the object to its classes, as shown in the figure 2.1.1.3. They are also used to describe the relation between a method itself and its executions, which is a fundamental step to simulate simple concepts like method calls or recursion.

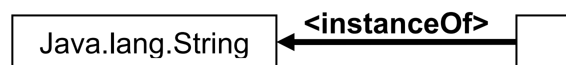


Figure 2.1.1.3. instanceOf edge example

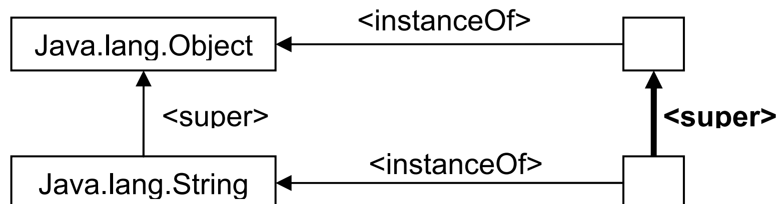


Figure 2.1.1.4. Super object edge example

The method declared edge is a link between a method and its defining class. This establishes the interface of each class. For example, the following method would be described by the figure 2.1.1.5.

```
public class Object(){
        …..
        String toString(){….}
}
```

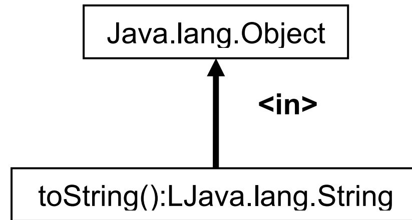


Figure 2.1.1.5. Method declared edge example

The object name edge permits naming objects. It is like a pointer that only allows method or assignment transformations. See figure 2.1.1.6.

Figure 2.1.1.6. Object name edge example

The method active edge indicates the method or statement that is being executed in a certain moment. It is created in the method / statement call (by the caller) and destructed in the method return / end statement (by the called). There is an example of this edge in the figure 2.1.1.7.

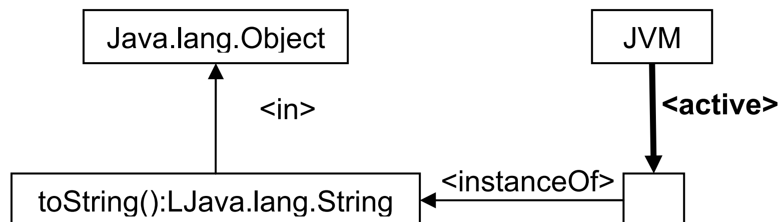


Figure 2.1.1.7. Active / scope edge example

The method caller edge gives to the called method instance the identity of its caller in order to provide its response (a return or a throw edge to the corresponding result of the method execution). This is illustrated in figure 2.1.1.8 .If a return/throw statement appears in the source code method body, a new <return> or <throw> edge from the active method instance pointing to the returned object (an object instance of a subclass of Java.lang.Throwable or an object instance of any other class) is created.

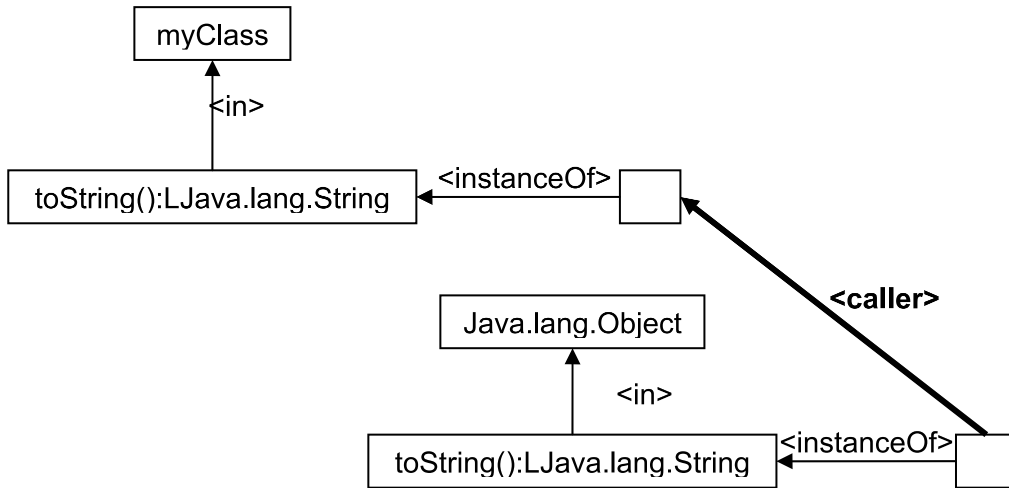


Figure 2.1.1.8. Caller edge example

When the last instruction of the method is executed all method instance relations are erased. If there is a returned object (an object instance connected from the method instance called by a <return> or <throw> edge), the method instance node will be replaced by the returned object. If there is not any returned object a void instance will be created to replace the method instance called.

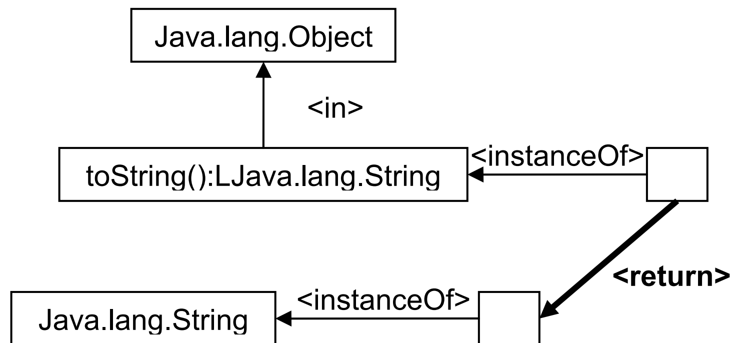


Figure 2.1.1.9. Caller edge example

Furthermore, a new temporal variable edge (<0>) will be created from the method instance caller to the returned object and a new active edge (<active>) to the method instance caller. See figure 2.1.1.10.

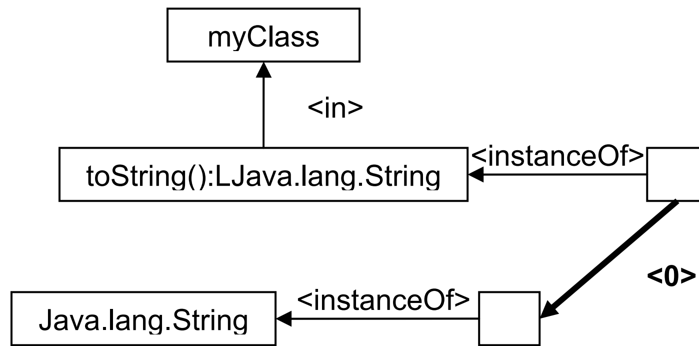


Figure 2.1.1.10. Temporal variable edge example

The loaded class edge serves as connector between the JVM node and loaded class nodes. This edge representation is displayed in figure 2.1.1.11.In this way, if the JVM has not loaded any referenced class, its load production rule will be the only one that matches at the moment that this class is needed. Besides, it also can contribute for marking all referenced objects starting from the JVM node. The non marked objects are not referenced anymore and should be deleted. Thus the class edge permits to simulate garbage collection.

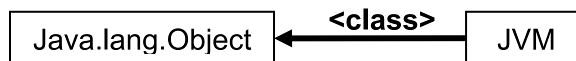


Figure 2.1.1.11. Loaded class edge example

The actual instruction (PC) edge is a link between a method instance and its corresponding index of instruction, see figure 2.1.1.12. Each method instance has its own order of instruction to simulate its state.

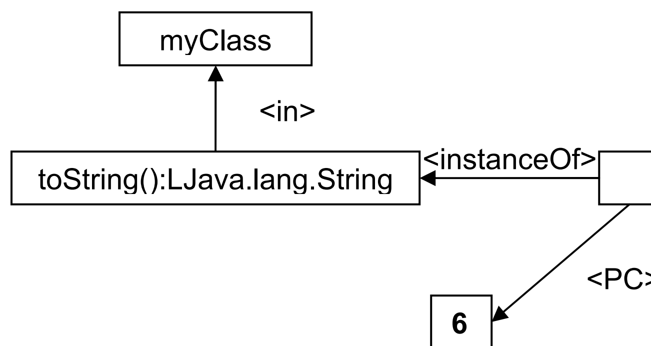


Figure 2.1.1.12. Index of instruction edge example

The temporal variable edges are objects defined without any identifier. This edge links them to their parent scope, in other words to the method instance in which they were created.

The operation result edge is an object name edge (named <0>) that has the result of executing an operation or statement. This means that they can point to the operation or statement node or to its result.

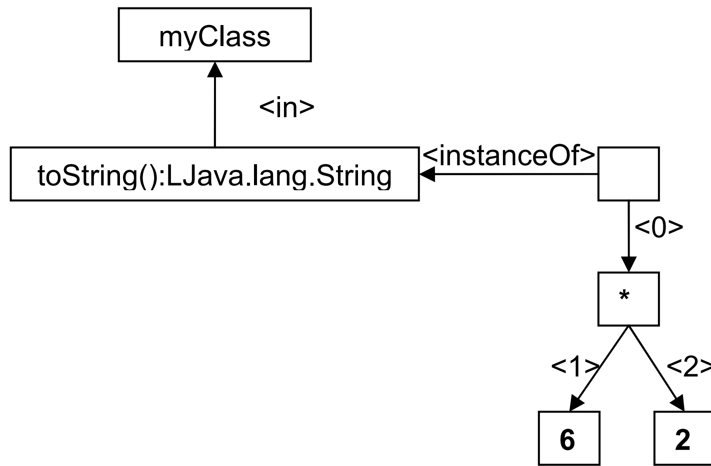


Figure 2.1.1.13. Operands edge example

The parameter edges are object name edges that correspond to each one of the parameters of a method instance, an operation or a control statement. They are named with the order they have in the caller definition (with positive numbers between <> starting with one).

The edges definition table condenses the types of edges that symbolize relations of object model entities in a Java environment. The first column enumerates the relationships among the represented software units (method calls, inheritance, membership, etc). The second shows its corresponding labels. And the third column has an abbreviation for each type of edge to generate a canonical textual representation of the defined graphs.

Type	Identifier	Description
Super class Super instance	–<super>→	**O_1** –<super>→ **O_2** **C_1** –<super>→ **C_2**
Instance of	–<instanceOf>→	**O** –<instanceOf>→ **C** **MI** – <instanceOf>→ **M**
Method declared	–<in>→	**M** –<in>→ **C**
Object name	–name→	**O_1** –name→ **O_2** **MI** –name→ **O_2**
Super interface	–<implements>→	**C_1** –<implements>→ **I_1**
Active / scope	–<active>→	**JVM** –<active>→ **MI_1** **JVM** –<active>→ **S_1**
Method caller	–<caller>→	**MI_2** –<caller>→ **MI_1**
Normal return	–<return>→	**MI_2** –<return>→ **MI_1**
Exception return	–<throw>→	**MI_2** –<throw>→ **MI_1**
Loaded class	–<class>→	**JVM** –<class>→ **C**
Actual instruction	–<PC>→	**MI** –<PC>→ **IO**
Index of instruction	– positive int → –positive int $ positive int→	**IO** –positive int →**IO** **IO** –positive int $ positive int→**IO**
Operation result / temporal variable	–<0>→	**MI** –<0>→ **OP** **MI** –<0>→ **S** **MI** –<0>→ **O**
Non named variables (Literals)	–value→	**O** –<value>→ **O**
Parameters	–<1>→ (first parameter) –<2>→(second parameter) –<3>→(third parameter) …	**OP** –< positive integer >→ **O** **MI** –< positive integer >→ **O** **S** –< positive integer >→ **O** **S_1** –< positive integer >→ **S_2**

Table 2.1.2.1. Edges definition

Note that the edges artificially created in order to add object oriented semantics to the graph are enclosed by angled brackets (<>), the edges whose label is not surrounded by brackets are explicit relations in the source code like the variable

names. The only exception is the index of instruction which identifies uniquely and adds ordering information.

By composition of these relations plus the set of nodes we can represent with a flat multipurpose graph Object Oriented entities in a Java approach.

2.1.3. Object oriented concepts as graphs

Some elements are defined with other elements depending on its defining relations like aggregation, inheritance, dependency, etc. For example, the representation of an object is the representation of its class (package, methods and super class –with its package, methods and super class-) and the representation of the object itself (object node and its super object node –which can also have super object node-).

A class is composed of the class node which is the main node, a super class edge pointing to its parent class node, a set of super interface edges (<implements>) pointing to its parent interface nodes, its method nodes connected to the class node with a method declared edge (<in>).

An object is defined by the object node, linked to its class reference by an instanceOf edge (<instanceOf>) and to its super reference by a super edge (<super>). The class fields are represented as links from the object node to the values labeled with the name of the field (fieldName).

A method is constituted by the method node linked to its class. A method cannot be linked to two different classes, the declared in edge (<in>) means that this method is defined in the pointed class. For example, if there is method overloading, there will be a method node per declaring class.

A method instance is formed by its method instance node linked to its method by an instance edge (<instanceOf>). When the method instance is called is created its caller edge (<caller>) pointing to the method instance that called the method, then, its this edge (<this>) is created pointing to the object that invokes the method call. If there is not an invoker object it is assumed that the actual object instance is the one that calls the method, in such case the invoker object is the same pointed by the this edge in the caller method instance.

The local variables (those ones created inside a method execution) are named using a name edge (name). The parameters edges (<number>) point to the objects sent to the method, in which the number indicates the order in which they are expected. Remark: in this version, the Translator only represents sending parameters by reference. Once these information requirements are fulfilled, the method instance is ready to start its execution.
Finally, so is its actual instruction edge (<PC>) will point to an instruction operation node labeled with a index of instruction zero (0). Also the <active> edge that is created, it indicates that the instructions of this method are those that will be applied, this means that the active edge actualizes the scope.

When the method call ends, the caller edge is used with return purposes, this means, a return edge (<throw> or <return>) is created from the caller pointed node to the returned object node.

The information obtained as a result of a production rule application is execution time information; it can be simulated by applying a production rule that clean up the environment created by the previous one but the applications of these rules are transparent for the user. For more detail see end production rules examples in section 2.4.

2.2. Class graphs

A class graph is the name given to a graph that represents any possible reference to a given class; this means, its fields, its methods and its super class.
This is achieved by modeling in a generic way every relation of a class. Any class is modeled with 3 basic types of relations: the methods that define (<in>), the fields that comprise it (name) and all its super types[7] (<super>).
The class graph comprises the object and class level, each class node has its corresponding object instance and each method node has its corresponding method instance.

For example, the class graph for C2 would be [8]:

```
package p1;
public class C2 extends C1{
```

[7] Chain of classes that extend.
[8] Note that all the source code in the example has public modifier, this is done because the model currently does not support modifiers.

```
        public C3 myField;
        public C2(){}
        public void m1(){}
    }

    package p1;
    public class C1{
        public C1(){}
        public void m1(){}
    }

    package p1;
    public class C3{
        public C3(){}
    }
```

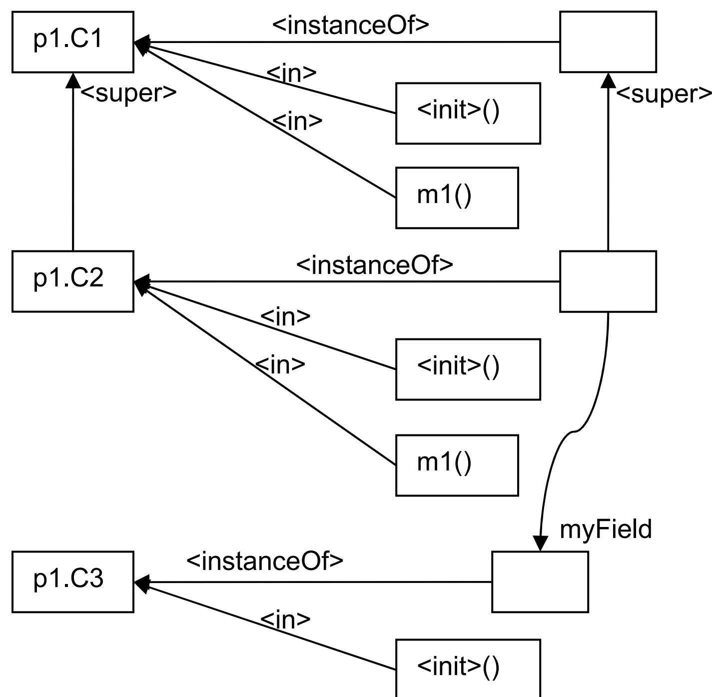


Figure 2.2.1. Class graph for class C2

The class C2 is composed of the class node (p1.C2), a super class edge pointing to its parent class node (p1.C1), an empty set of super interface edges pointing to its parent interface nodes and its method nodes connected to the class node with a method declared edge -C2(), m1()-.

The object instance of C2 is defined by the object node, linked to its class reference by an instanceOf edge (p1.C2) and to its super reference by a super edge. The class fields are represented as links from the object node to the values labeled with the name of the field (myfield which is a C3 instance).

2.3. Object graphs

The object graphs are instances of class graphs. Essentially they are class graphs with values for the instances of the primitive classes; they are objects in a given execution time (with a specific **IO** value).

For example, the class graph for C2 would be:

```
package p1;
public class Main{
     static public void main(String[] args){
          C2 c = new C2();
          c.m1();
     }
}

package p1;
public class C2 extends C1{
     public C3 myField;
     public C2(){}
     public void m1(){}
}

package p1;
public class C1{
     public C1(){}
     public void m1(){}
}

package p1;
public class C3{
     public C3(){}
}
```

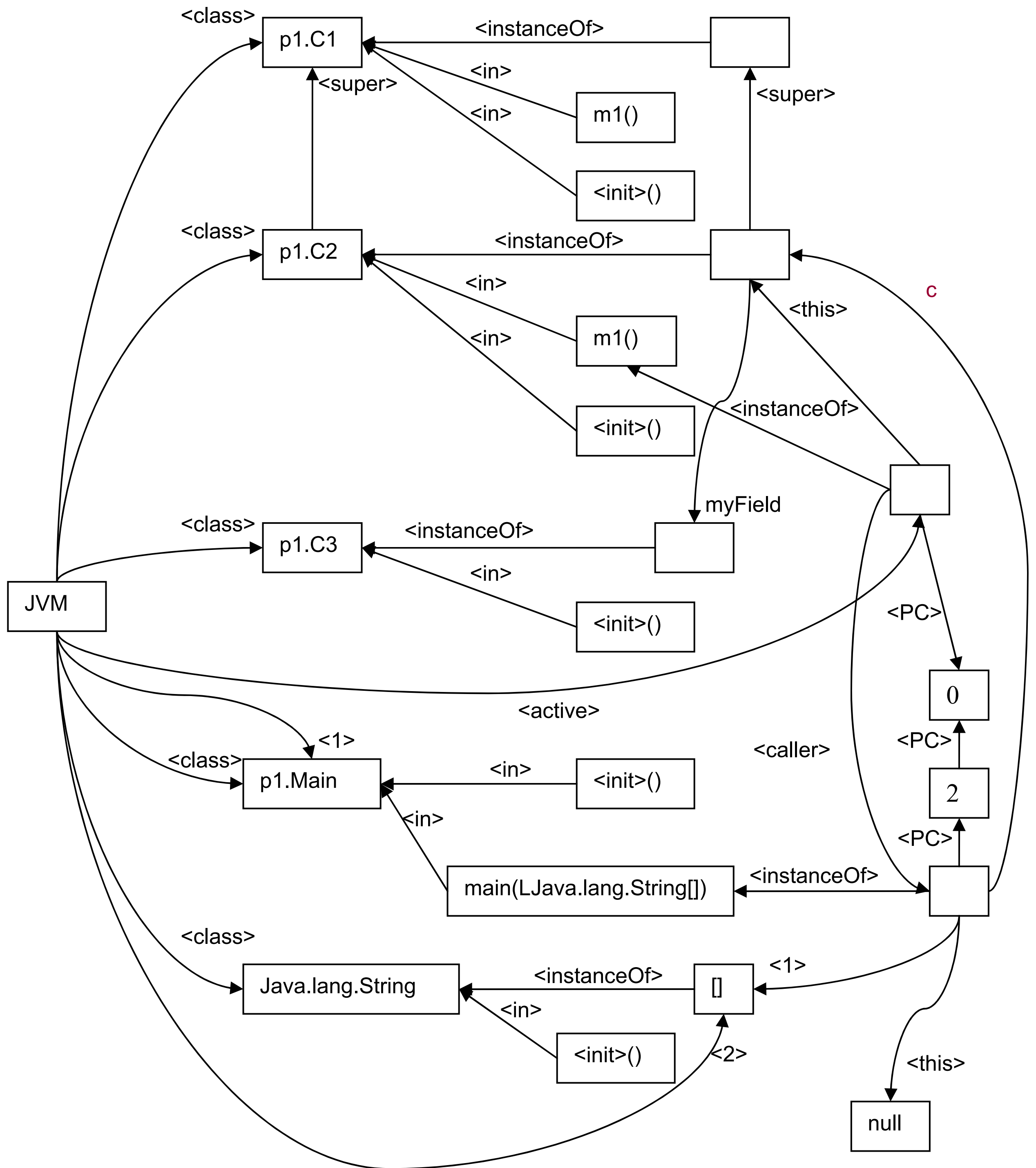


Figure 2.3.1. Object graph in c.m1();

2.4. Pattern rules

The semantic established by a code transformation can be represented as a production rule. This generic production rule can be 'instantiated' by filling its free variables with the matching equivalents of any object.
This section explains the principal contribution of this thesis, which is the translation into production rules of the semantic of Java statements.

The statements chosen were: for representing the branch the *if* statement; for representing the loop the *while* statement; for representing method interactions the method call and return and for representing object transformations: assignment, *operators*, *variable* and *field declarations*. This set of statements aim to be a minimal to reproduce basic Java functionality and to proof that it can be represented without loss of information as graph transformations.

The following sections will explain in detail how the semantic of these statements was model into rule patterns.

2.4.1. Literal creation

A literal is an object in particular a primitive class instance that does not have name. For example, `5`, `2.5` or `"hello world!"`.
They are differentiated by their value. Furthermore, their value is not only necessary to distinguish between them but also the allow operators execution.
As they do not have explicit creation statements it is compulsory to create them each time they are mentioned, otherwise the following instruction will not find its entire required elements.

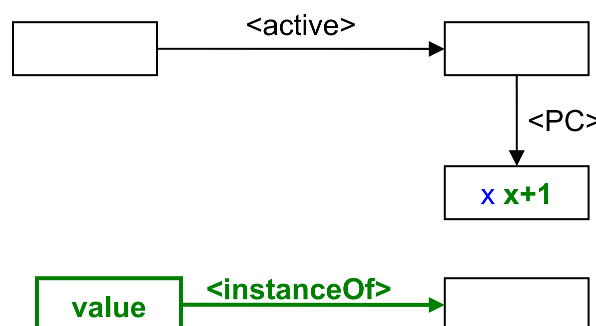


Figure 2.4.1.1. Literal creation generic production rule

To create a literal is necessary to know its value, its type node, and the active node. As the figure 2.4.1.2 depicts a new node is created in order to represent the literal object, then it is added a self referenced edge labeled with the literal value and an *instanceOf* edge from the new node to the type node. Finally the *index of instruction* edge is replaced by another labeled with the next integer value.

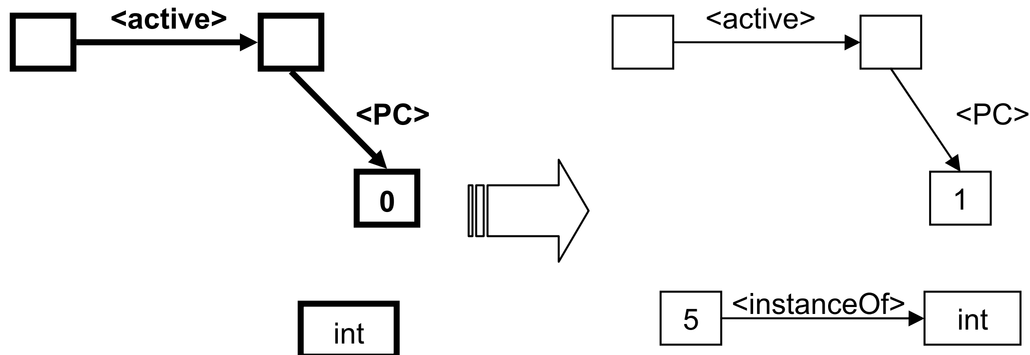


Figure 2.4.1.2. Literal creation production rule application example

The example in the figure 2.4.1.2 shows the result after applying the literal creation rule. In this case, was created a 5, and the actual instruction was incremented by one. The narrowed elements in the initial graph mark the corresponding match to the rule shown in the figure 2.4.1.1.

2.4.2. Variable declaration

A variable is a named object only can be accessible inside a well defined scope. The scope is given by the method or statement in which the variable was created. To represent the scope, variables are created at the in the instruction order indicated by the source code and destroyed with their scope. This representation allows in a nested scope to access the parent scope variables, but a high scope will not attempt to refer a internal scope variables because the inner scope will be executed and deleted once it instructions are finished.

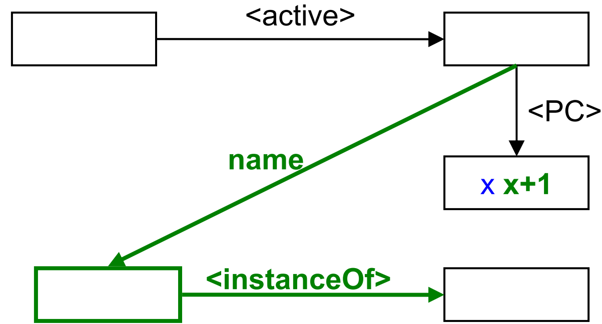


Figure 2.4.2.1. Variable declaration generic production rule

The actual scope is represented by the active edge that explicitly points to the method instance or statement whose instruction is executing in a given moment. The variable name is the label of an edge that links the scope node to the new object node.

As illustrates the figure 2.4.2.1, creating a new variable means create the node that represent as object, create its name edge form the scope node to the variable node and a *instanceOf* edge from the variable node to its type node. Then replacing the *index of instruction* by another edge labeled with the next integer value.

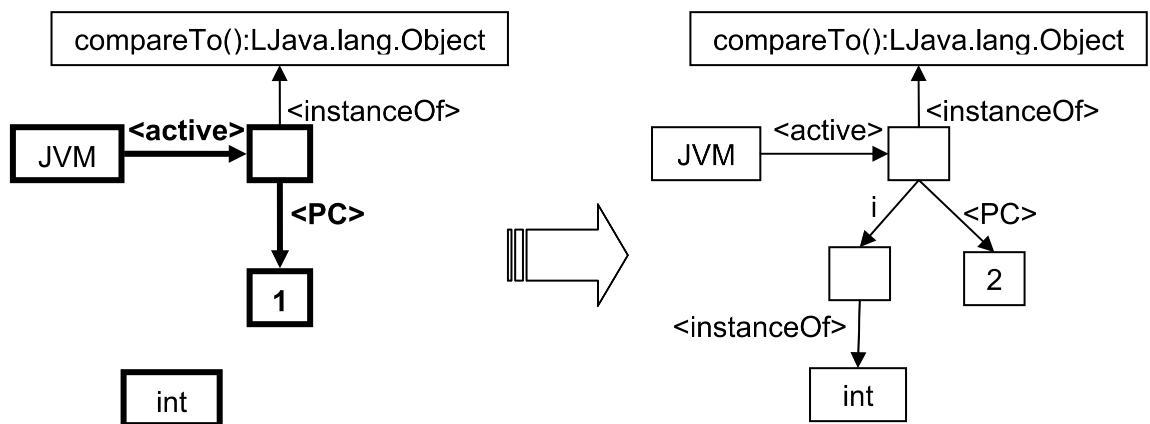


Figure 2.4.2.2. Variable declaration production rule application example

The example in the figure 2.4.2.2 shows the result after applying the variable declaration rule. In this case, a new variable i of type int was created inside the method compareTo, and the actual instruction was incremented by one. The narrowed elements in the initial graph mark the corresponding match to the rule shown in the figure 2.4.2.1.

2.4.3. Operations

An operator performs an action on one or two operands. As result of performing the specified action, an operator can be said to return a value (or evaluate to a value) of a given type. The type depends on the operator and the type of the operands. To evaluate to a value means that after the action is performed, the operator and its operands are effectively replaced in the expression by the value that is returned.

The operations are operator's actions, they are modeled as a node that represents the operator, and in order to be able to apply them unambiguously they are self referenced with the operator and a reference to its operands: the first one labeled with a <1> and the second one labeled with a <2>. With the purpose of leaving the value obtained as the operation result, the scope node (the method or statement active) is connected to the operator node with an edge labeled <0>.

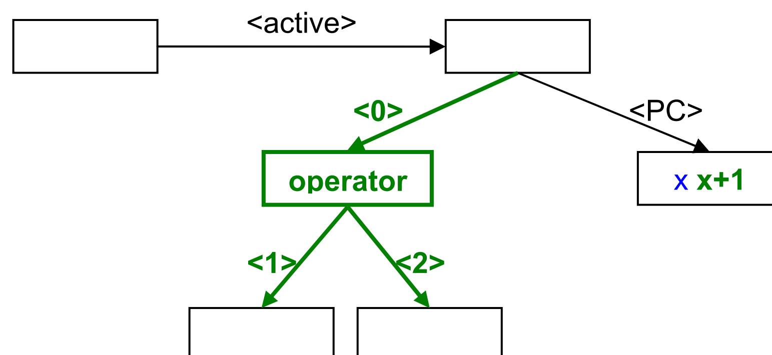


Figure 2.4.3.1. Operations generic production rule

The figure 2.4.3.1. presents a general operation production rule. It is possible to appreciate the prerequisites to create them: the scope node (method or statement active), the index of instruction edge (in this case represented with an x) and the operand nodes. With them the production rule construction consists in creating the operator node and self reference (labeled with the operand), the operand edges (labeled <1> and <2> respectively) and the result edge (labeled <0>). Finally, the value in the label of the instruction edge is incremented by one to indicate that the operation call was completed.

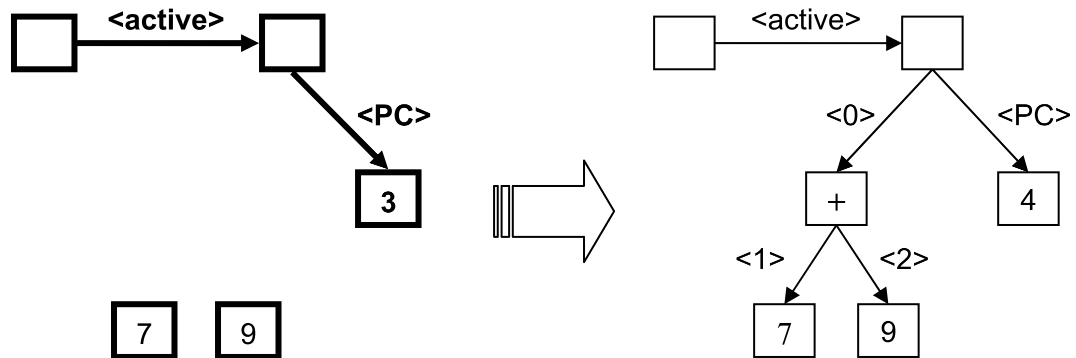


Figure 2.4.3.2. Operations production rule application example

The figure 2.4.3.2 demonstrates how is created a new operation call. The narrowed elements in the initial graph mark the corresponding match to the rule shown in the figure 2.4.3.1..

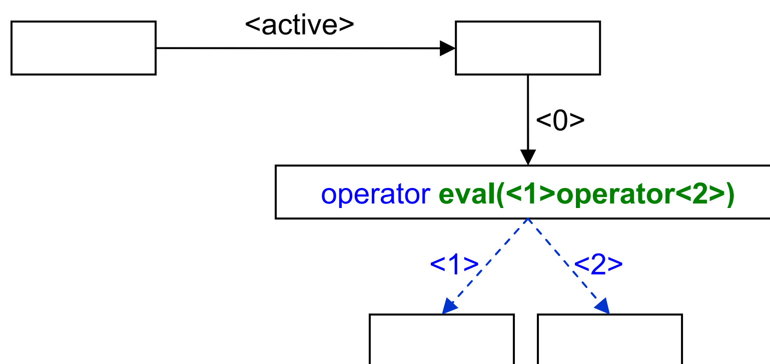


Figure 2.4.3.3. End operation production rule

Once the operation is executed the operands edges (labeled with <1> and <2>) and the operator self reference will be deleted. The operator node will acquire a new self reference labeled with the literal value obtained from the operation evaluation. The picture above presents the result of executing an operation.

This evaluation is done automatically by the simulator that is in charge of evaluating the operation and creation of the literal obtained as result. Nevertheless, it must be simulated somehow by the Translator in order to maintain the object graph consistent for the creation of the next instruction, for that reason, and given that it will not always have the values of the operands; it limits to delete the operator and operands nodes and leave the object graph with the representation of the operation result but without its value.

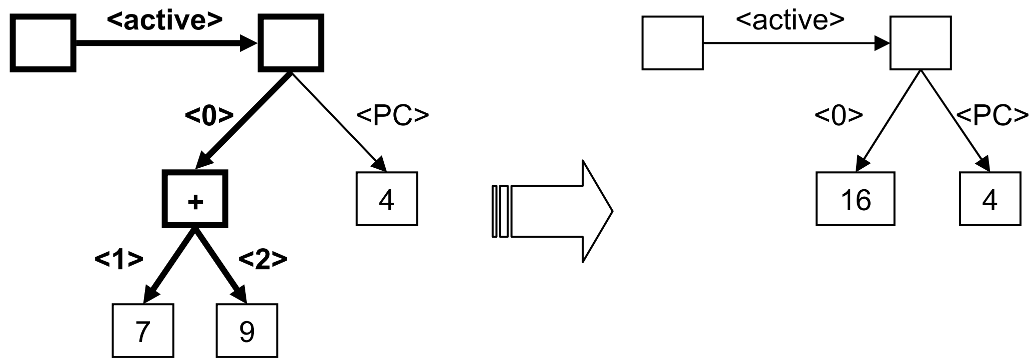


Figure 2.4.3.4. Operations production rule application example

The figure 2.4.3.4. exemplifies how an operation application would see in the simulator. The narrowed elements in the initial graph mark the corresponding match to the rule shown in the figure 2.4.3.3.

2.4.4. Assignment

The assignment is an operation that modifies the value of the left hand side object with the value of the right hand side object. The right side of an assignment expression is always known because it is evaluated before the assignment takes place.

The assignment is represented as a change of pointed value in a given variable. This is accomplished deleting the variable edge and creating a new one exactly as the deleted one except for the target value that points to the correct side object node. As any operation the assignment modifies the instruction edge incrementing its value by one to express that the assignment is done. This graph transformation is shown in the graphic below.

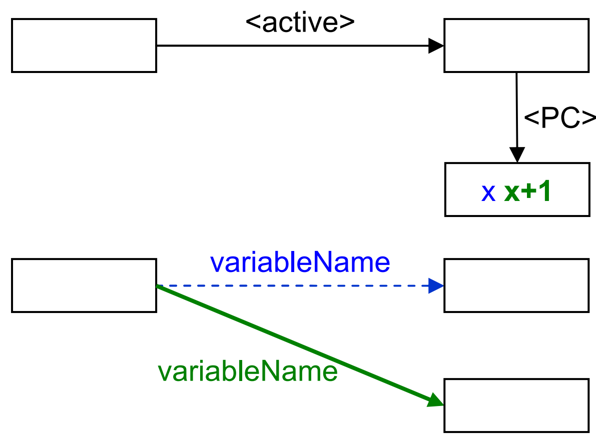


Figure 2.4.4.1. Assignment generic production rule

The figure 2.4.4.2 shows an assignment execution, the narrowed elements in the left hand side graph are the ones that match the production rule of the figure 2.4.1.1. As shown in the example, the initial node value of the variable is not deleted in the production rule for two reasons: it does not occur when an assignment takes place and it could have other objects that reference it. The non referenced objects are supposed to be deleted with another production rule that would simulate the execution of a garbage collector.

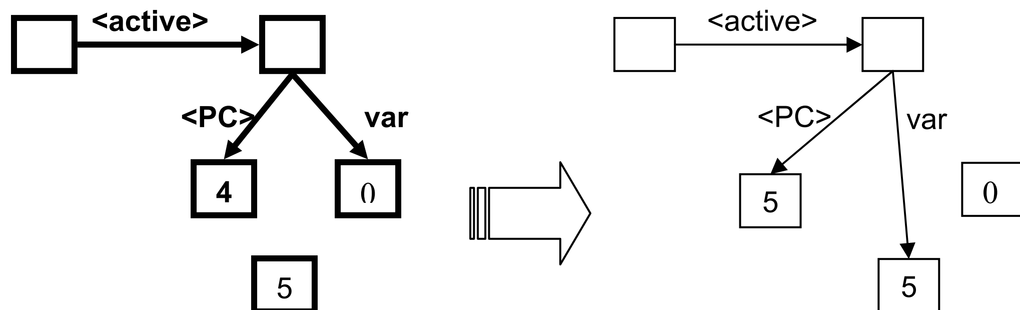


Figure 2.4.4.2. Assignment production rule application example

2.4.5. Methods

Methods define how an object responds to a message or request. Each method has its own scope, for that reason each time a method is called a new method instance is created with some requires variables in order to be able to be executed.

Consider: variable = object.method(variable2);

The following process takes place:

1. Find object pointed to by object, let's call it invoker.
2. Find code for that object's class
3. Find code for method()
4. Find object pointed to by variable2, this is the parameter passed to the method execution
5. Run code
6. Return value
7. Use returned value and assign it to variable

From the first to the fourth steps allow creating a new one for the new method instance, we will call it method creation environment and saving the actual state of the execution to be able to return to the previous scope once the method execution has finished. To execute the fifth step is necessary to effectively call the method. Finally, a method can have a return statement that indicates to the caller object which object is the result of the request. In some cases the method just changes the internal state of the invoker and does not need to return any object, in those cases, it returns a void object. The return statement execution is called here method return.

As these macro steps of the method call are carried out in the following order:

1. Method environment creation
2. Method call
    a. Method execution
3. Method return
4. End method call

The method execution assumes that the method called has pre established its parameters, and environment references and just executes as any instruction.

The end method call performs at once: the destruction of the method environment, the return of the control to the caller scope and the storage of the returned object in a non named value.

2.4.5.1. Method environment creation

The method environment creation rule creates a new method instance given its invoker object node, its method node, the active node (in order to save the previous execution environment) and its parameters.

With those elements it creates the new instance method node, connected to the previous scope node by a caller edge, to the invoker object by a *this* edge, to its method node by an instanceOf edge and to its parameters by edges labeled with the expected order of parameters. To indicate that the method environment creation was finished, it also increments by one the instruction edge, as illustrates the figure 2.4.5.1.1.

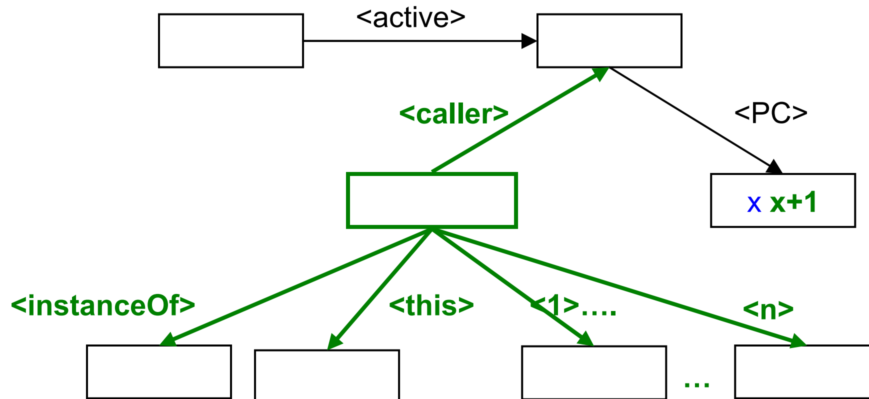


Figure 2.4.5.1.1. Method environment creation generic rule production

The generic production rule knows that the number of parameters is variable, i.e. a method may not have any parameters. For this reason, the specific production rule may or may not have these edges. The figure 2.4.5.1.2.a depicts a method creation environment production rule of a method with one parameter.

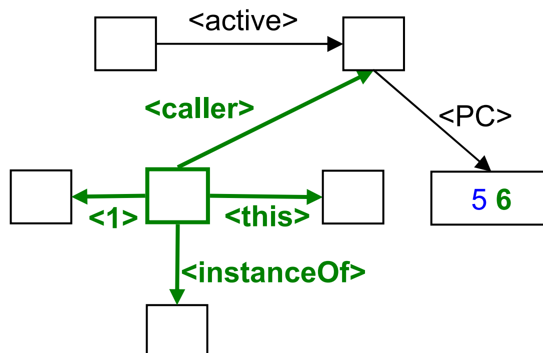


Figure 2.4.5.1.2.a. Method environment creation production rule example

The picture 2.4.5.1.2.b. shows a method environment creation execution, the narrowed elements in the left hand side graph are the ones that match the production rule of the figure. 2.4.5.1.2.a

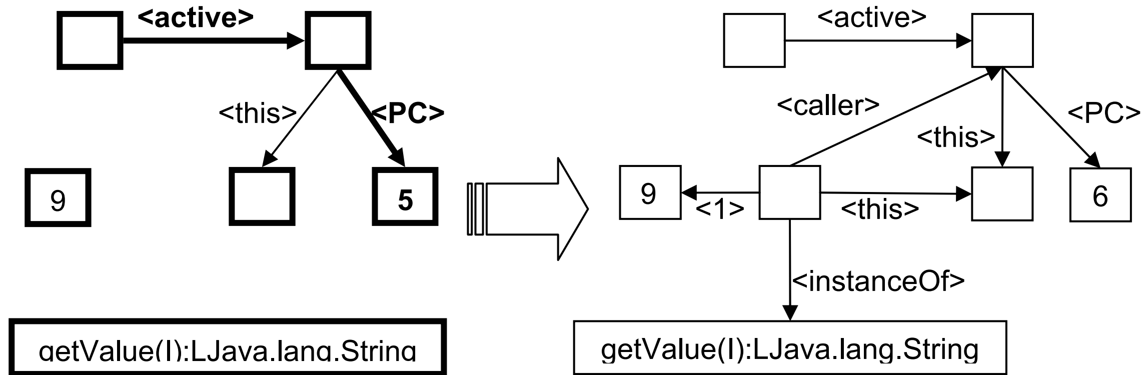


Figure 2.4.5.1.2.b. Method environment creation production rule application example

2.4.5.2. Method call

The method call effectively activates the new method instance by giving to it two essential elements to, its own instruction index node with an initial instruction zero and activates its scope with the active edge. In order to be able to construct this production rule is necessary to have the active edge, the new method instance and the instruction index node of the actual scope node. The generated production rule is depicted in the following illustration.

The creation of a new instruction index node can also be called as PC nesting because the new instruction index depends of the previous one by a PC edge.

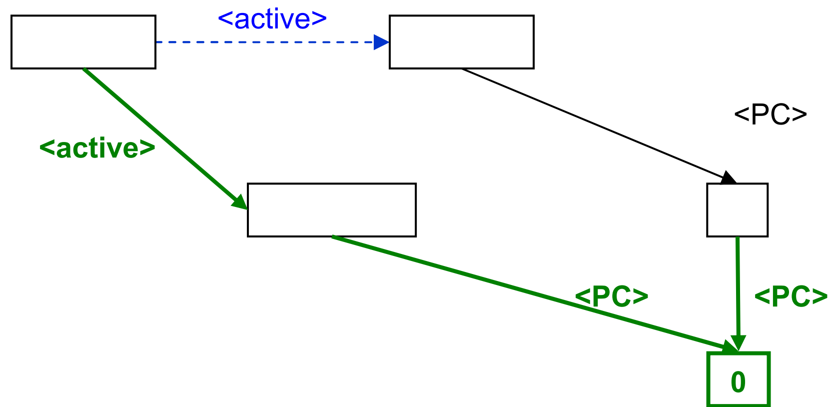


Figure 2.4.5.2.1. Method call generic rule production

The figure 2.4.5.2.2 illustrates how a specific method call production rule match in a graph and which is the resulting graph once the rule is applied. The elements in the left hand side graph irrelevant for the rule application are not narrowed.

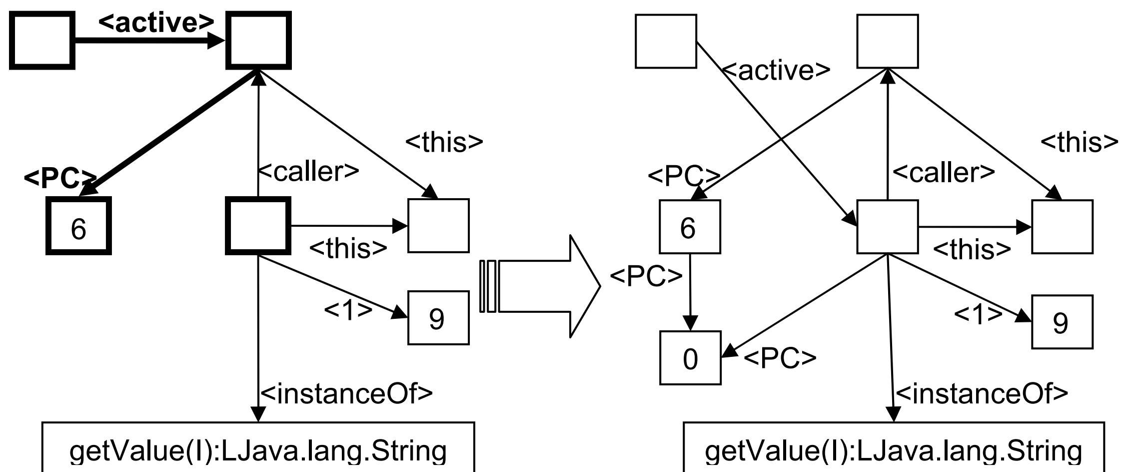


Figure 2.4.5.2.2. Method call production rule application example

2.4.5.3. Method return

Some methods contain a return expression. However, the value-returning methods as the void methods will have a return edge that explicitly points to the returned object or to a void object if it does not have return statement.

To create a method return production rule is necessary to have the returned object node, the method instance node (is also the active scope node), and the

instruction edge. With these elements the production rule is generated by creating a return edge from the method instance node to the returned object node, and replacing the value of the instruction edge label for the next one. Note that replacing a label of an edge or its source or target nodes implies delete the existing edge and creating a new one with the desired elements because the edge identity is given by its set of components.

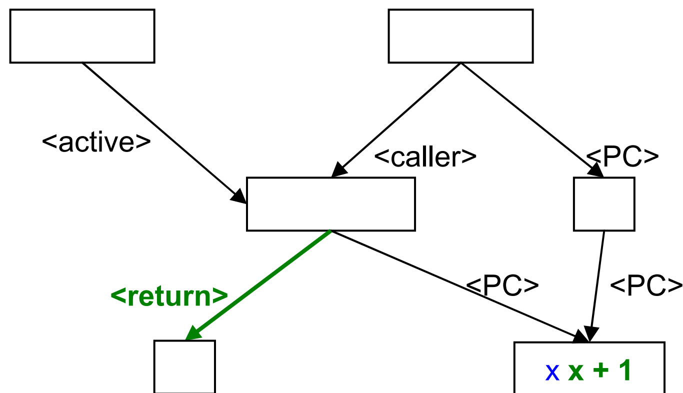


Figure 2.4.5.3.1. Method return generic rule production

The figure 2.4.5.3.2 depicts the application of a specific method return production rule. The narrowed elements are those that match with the rule, what means that they are required for applying the rule.

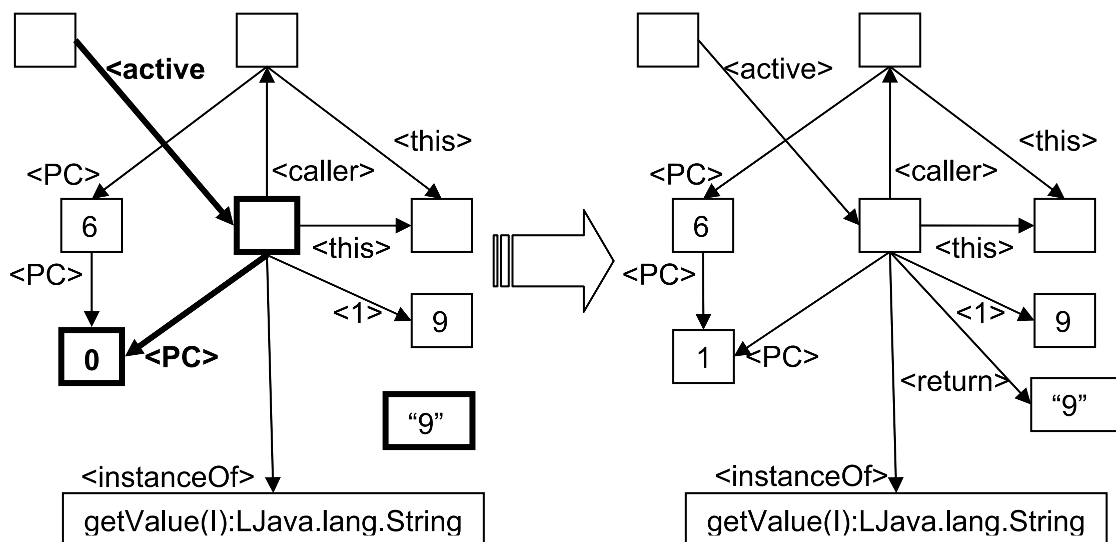


Figure 2.4.5.3.2. Method return production rule application example

Once the method execution has ended the method environment must be deleted in order to return to the execution point in which the method was invoked. In other words, the method instance and all its required references (caller, method, invoker object, parameters and instruction index) are deleted. The node pointed by the return edge is linked to the caller scope node as a non named / temporal variable, and the active edge leave of pointing the method instance to point to its caller scope. This can be seen in the figure 2.4.5.1.

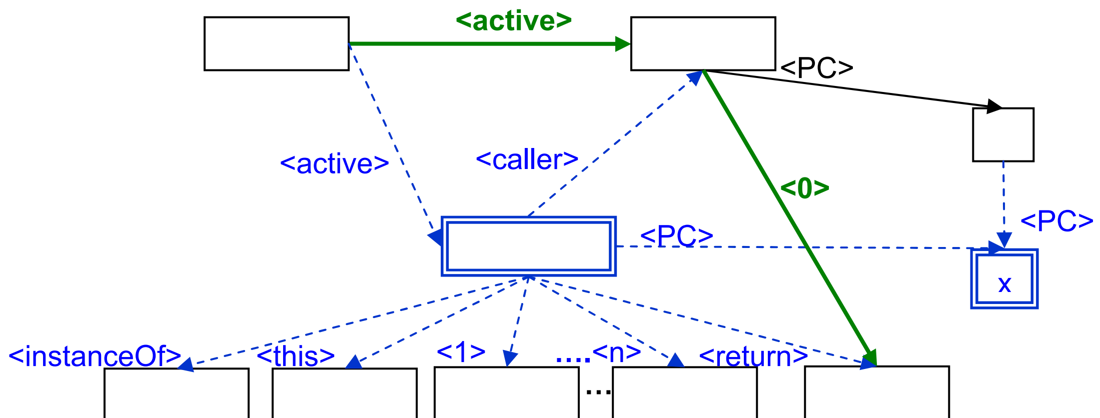


Figure 2.4.5.1. End method environment creation and method call production rule

2.4.6. If

The representation of if statement is solved in 3 steps:

1. If condition
2. Then (or true branch of if
3. Else (false if branch)

As if condition is an operation result, it is evaluated before of any other part of the if statement. Once the condition is evaluated the if node is created in order to save the value of the condition evaluation and associate it to the if statement.

Note that any type of if arrangement can be defined with this if-then-else basic representation. For example, consider:

```
if (bool){
      variable = 0;
}
else if (variable > 0){
      variable = variable * -1;
```

```
}
else{
     variable = variable + 1;
}
```

the previous code would be represented in the following way:

```
if (bool)
     then
          variable = 0;
     end then
     else
          if (variable > 0)
               then
                    variable = variable * -1;
               end then
               else
                    variable = variable + 1;

               end else
          end if
     end else
end if
```

Once the if node is created is necessary to create the transformation rules in the event that the condition had been evaluated as true or as false. Given that both then clause and else clause may have a dependent statement block, they must have a differentiating instruction edge that permits the execution of many instructions. Besides, they require a unambiguously instruction value which explicitly state in which branch of the if statement the instruction is situated. These two instruction conditions are accomplished by adding some special structure to this kind of statement nesting (statements inside the then-else statements). The first token would represent the if or parent statement, the second one would represent the value of the branch condition and the last one the index of the next instruction. For example the instruction index 8$1$0 would mean that the following instruction is part of the if statement whose instruction index is 8, is an instruction of the then clause because it is followed by the 1 (in this case it represents true condition) and is the first instruction (ends with a 0) of

the statement block of the then part in the if statement. An instruction index 8$0$2 would mean it is the third instruction of the else branch of the if statement whose instruction edge label is 8.

These special instruction indexes have two advantages: they allow a unique instruction naming representation and they permit the exact execution point of any nested instruction.

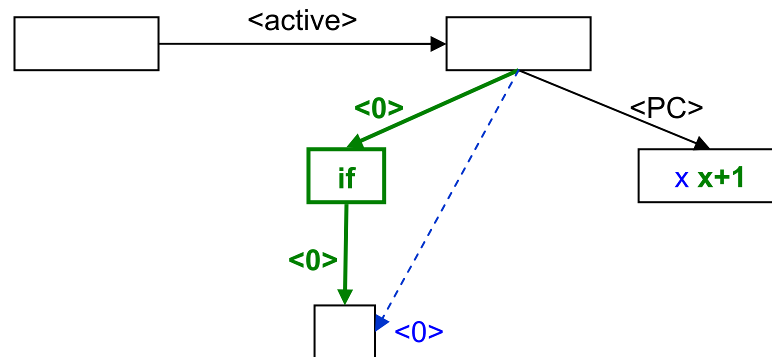


Figure 2.4.6.1. If generic production rule

The figure 2.4.6.1 depicts a new if node creation. To be able to generate it, it is necessary to have the active node that defines the scope in which if statement was created, the temporal variable that contains the value of the condition evaluation and the instruction edge that permits establishing the value of the next instruction. With these elements is created the new if node, its self reference that identifies it (if), and two temporal variable edges (those labeled with a <0>) the first one connects the active node to the new if node and the second one connects the if node to the condition value. As a new temporal variable edge is created from the active node to the if node, the previous one that pointed to the condition evaluation is deleted. Finally, the instruction index is replaced by another one with the previous label value incremented by one.

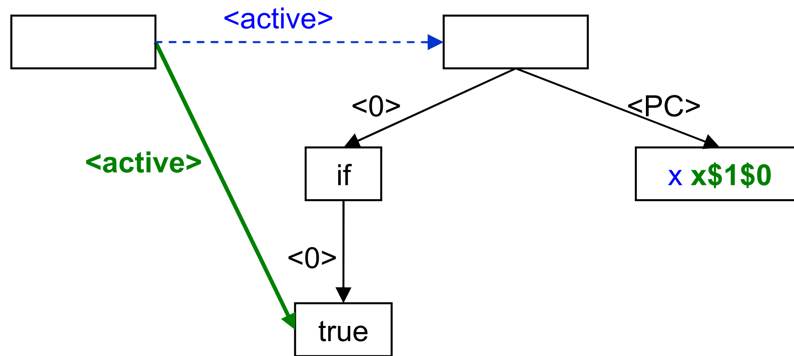


Figure 2.4.6.2. If true generic production rule

In the picture above there is a then clause creation, as it shows, there is a special environment creation for this new block statement. It means that there is instruction autonomy because there is a special set of instruction indexes for the block statement (from x$1$0 to x1n), it also means that it is a new scope block and for that reason the active edge now points to then node (a self referenced true node which is temporal variable of an if node).

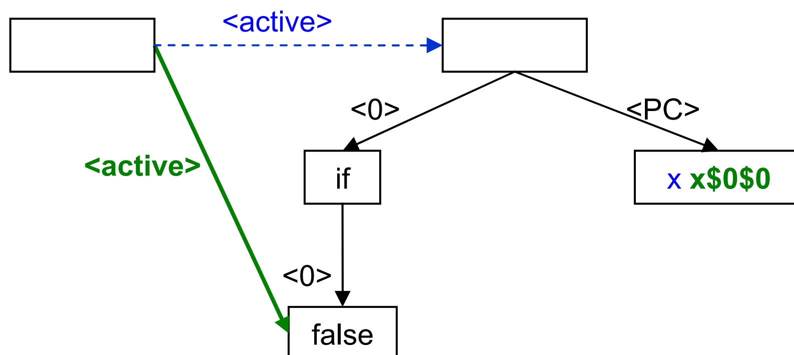


Figure 2.4.6.3. If false generic production rule

The figure 2.4.6.3 depicts an else clause creation, as the then clause there is a special instruction set of indexes and the active scope is transferred to the else node (a self referenced false node which is temporal variable of an if node).

In order to create these branch production rules (true/then and false/else) is necessary to know which is the corresponding if node, the active edge, and the instruction edge. As these rules are generated based on the source code the graph does not have the condition execution value. Given that if condition is a boolean value it is inserted artificially into the graph when the branch is created; as it has its own set of instruction values, the condition value is not needed

anymore inside the generation and application of the production rules of the instruction block.

The rule deletes the previous active edge and creates a new one with the same source node and label but with target the condition value node, the instruction index is replaced by an structured one depending on the branch created.

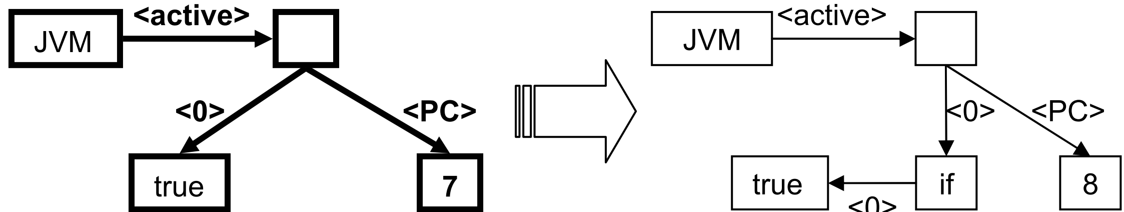


Figure 2.4.6.4. If production rule application example

The figure 2.4.6.4 shows an if statement creation. The narrowed elements on the left hand side graph are those that match for a specific if rule creation. The others are not required to apply the rule.

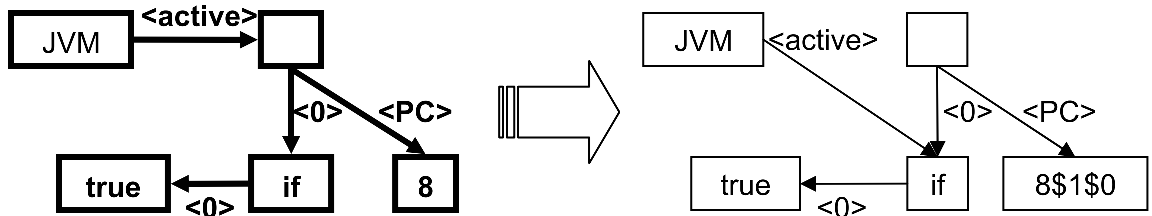


Figure 2.4.6.5. If branch production rule application example

The previous graphic illustrates a then (true branch) creation. The not narrowed elements on the left hand side graph are those irrelevant for the rule application.

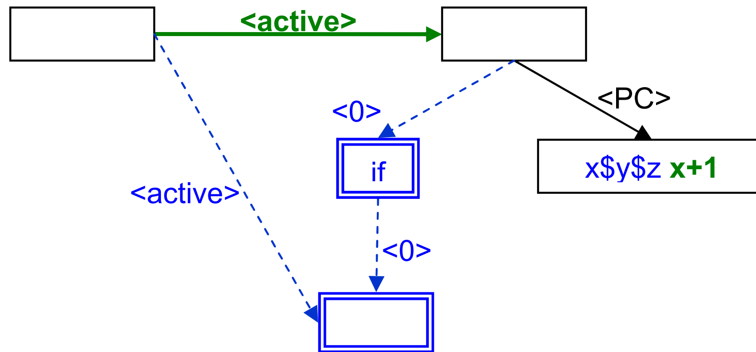


Figure 2.4.6.6. End if branch and if production rules

Once the branch set of instructions are translated the branch execution environment is destroyed, this means that the edge that connects the if node with its condition node is deleted, as well as the condition node. The previous scope (i.e. the node that called created the if node) is activated, deleting the active edge that was pointing to the branch node and creating a new one pointing to the previous scope node. The structure given to the instruction edge is deleted, remaining the value previous to the branch creation.

Then is deleted the if production rule by removing the if node, its self reference edge (if) and the temporal variable edge that linked the previous scope node to the if node. Finally, the instruction edge labeled is incremented to manifest that the end if was accomplished. This rule (as all ending production rules) is executed in a transparent way to the user in order to maintain the graph consistency along the transformation (i.e. to simulate the instructions execution) but is not part of the output transformation rules.

2.4.7. While

The while representation is a slightly different from an if representation, the main difference is the while semantics, also represent an execution split but with an instruction set that must be executed as many times as the condition complies whereas when the condition does not comply the while execution must be omitted.

Then the while is represented in four steps:

1. While condition
2. While instruction block (true condition)
3. While jump to next instruction (false condition)

4. While jump to condition (final while instruction)

The while condition is done exactly as the if condition the only difference is that the new node is a self referenced with a while label. This step is presented in the figure 2.4.7.1:

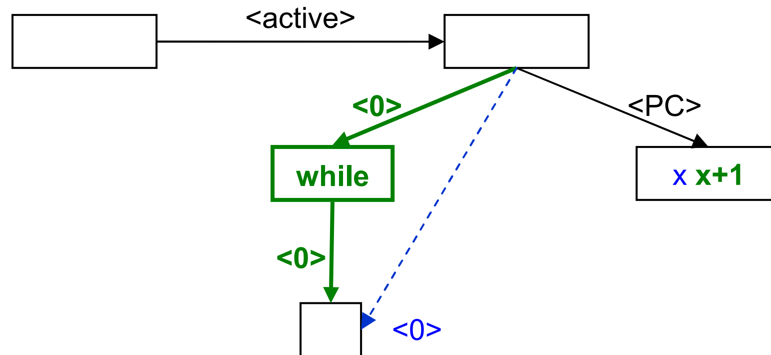


Figure 2.4.7.1. While condition generic production rule

Once the while node is generated the complying and not complying actions are generated.
When the while condition is set to true, the production rule creates the environment for the while instruction block. This implies to transfer the active node to the while condition node and structuring once its instruction edge. This is done because there is no need to distinguish between the execution when the condition is true and when it is false, because when it is false it just continues the previous scope instruction order. The figure 2.4.7.2 illustrates how a while true production rule is seen.

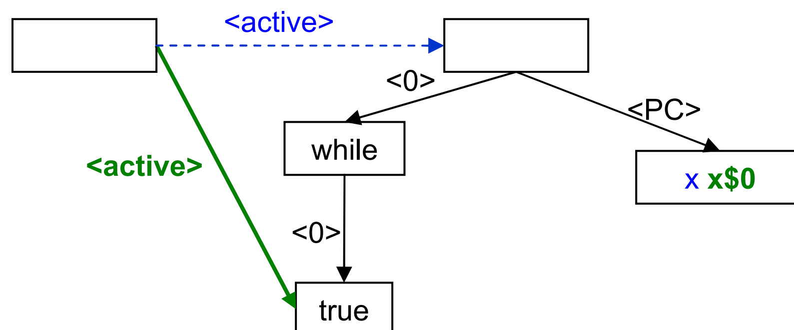


Figure 2.4.7.2. While true generic production rule

When the while condition is set to false, the rule created must destroy the while statement and jump out the while scope. In other words, it has to delete the while

and condition nodes and their relations, this includes self references (while and false), temporal variable edge that connects the active scope node and the while node and temporal variable edge that links the while node to its condition node. The instruction edge labeled is replaced with another with the next value, in order to jump to the instruction after the while statement. See figure 2.4.7.3.

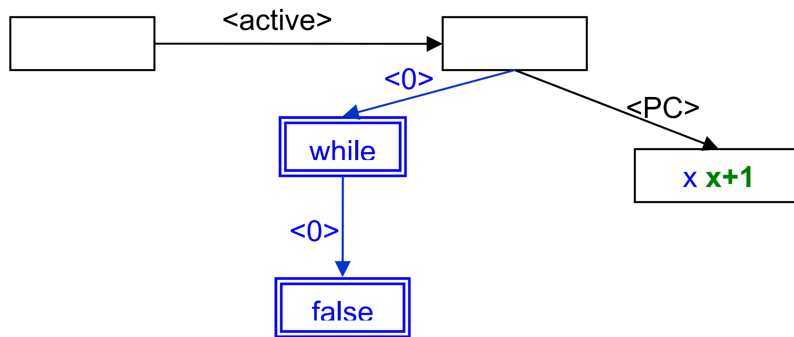


Figure 2.4.7.3. While false generic production rule

The next graphic displays a while statement creation. The narrowed elements on the left hand side graph are those that match for a specific if rule creation whose instruction edge is 10. The others elements (like the JVM edge) are not required to apply the rule.

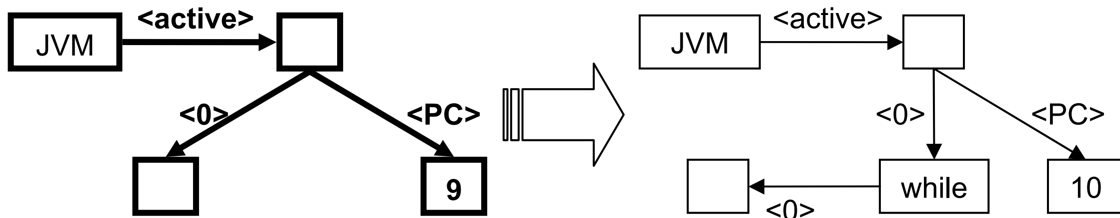


Figure 2.4.7.4. While production rule application example

The figure 2.4.7.5. shows a while instruction block creation. The not narrowed elements on the left hand side graph are those irrelevant for the rule application.

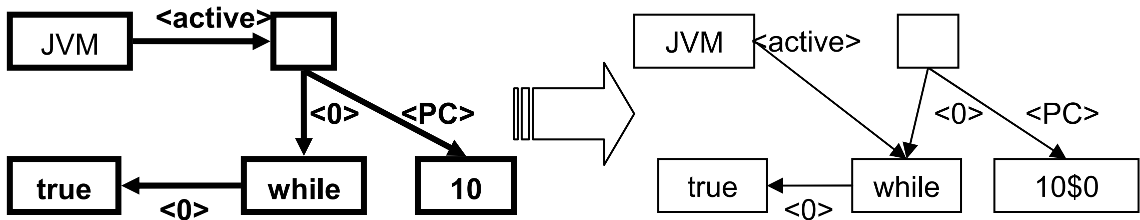


Figure 2.4.7.5. While true production rule application example

Once the instruction block is translated the true while branch execution environment is destroyed, this means that the edge that connects the while node with its condition node and the one that connects the previous scope (i.e. the node that called created the while node) to the while node are deleted, as well as the condition and while nodes. The previous scope is re-activated, deleting the active edge that was pointing to the condition node and creating a new one pointing to the previous scope node.

Finally, the instruction edge structure is eliminated and its obtained value is decremented to ensure that the last while block instruction will jump to reevaluate the condition.

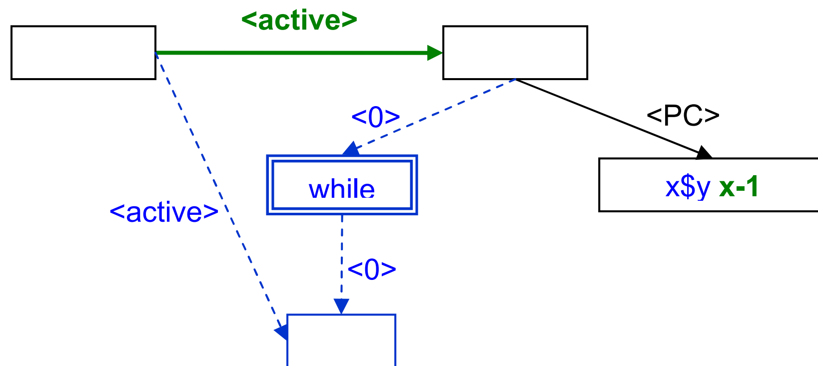


Figure 2.4.7.6. While jump to condition production rule

3. Tool Description

Translator is a software application developed using Java language. It is designed on top of GROOVE (GRaphs for Object Oriented Verification). To learn more about GROOVE, see [4].

This software is result of a graduate final project, carried out with TRESE, Computer Science, University of Twente, The Netherlands. Translator was developed by Angela Lozano (Software Designer and Programmer) and Arend Rensink (Project advisor and facilitator of Twente University).

This section explains how to use or extend the Translator application.

3.1. User guide

Translator allows you to translate from Java source code to graph transformation rules. By using Translator you can get a set of files that represent the semantics of a Java source code. With these files you can simulate the code execution. In a future version GROOVE will be able to verify the code consistency. Finally, Translator is useful in creating simple and intuitive graphical representations of Java programs.

This section is to provide guidance on using Translator tool.

3.1.1. System Requirements

As this software is developed using the Java language it requires its system specification. This specification is stated in Java documentation, for more information please visit: http://java.sun.com/j2se/1.4.1/install-windows.html if you have Windows as operating system, http://java.sun.com/j2se/1.4.1/install-linux.html if you have Linux as operating system and http://java.sun.com/j2se/1.4.1/install-solaris.html for Solaris machines.
Using a more powerful system will certainly enhance the software performance.

3.1.2. Installation Instructions

This program is built using Java Language; this means that you need the Java Development Kit, known as JDK/SDK/J2SDK. Before you run Translator, you must install the JDK 1.4 software in your pc.

The first step you have to do is get the software from:

http://java.sun.com/j2se/1.4.1/download.html

Next steps are explained here:

http://servlet.java.sun.com/help/installation/

Make sure JDK is properly installed and class paths are set. Details for Linux and Solaris operating systems see:

http://java.sun.com/j2se/1.4.1/docs/tooldocs/solaris/classpath.html

For windows operating systems please go to:

http://java.sun.com/j2se/1.4.1/docs/tooldocs/windows/classpath.html

Now that you have configured your JDK, you can proceed with the GROOVE installation, needed for viewing and simulating Translator files.

First, download the zip archive with the jar and batch files from:

http://www.cs.utwente.nl/~ GROOVE

Then, unzip and copy the jar (*.*jar*) and scripts/batch (.*bat* extensions are for Windows

batch files, extension-less files are Unix shell scripts) files to a suitable local or system-wide directory, e.g., (for Unix) $HOME/lib/GROOVE or (for Windows) "C:\Program Files\GROOVE"

Finally, modify the scripts/batch files so that they contain the correct references to the jar directory and an installed jdk bin directory (JDK 1.4). For instance,

set JDK="C:\Dev\j2sdk1.4.1"

set LIB_DIR="C:\Documents Program Files\GROOVE"

3.1.3. Instructions of use

The tool set comprises the following tools (.bat extensions are for Windows batch files, extension-less files are Unix shell scripts):

- Editor[.bat]: for editing graphs and graph production rules.
- Viewer[.bat]: for viewing existing graph production systems, i.e., collections of graph production rules.
- Simulator[.bat]: for simulating graph production systems, starting in a given initial graph.
- Translator[.bat]: for mapping form Java source code to production rules

To run the software in Windows, double click on the Translator.*bat* file in the batch files directory. In Unix execute the script Translator.

The application starts and the following window should appear.

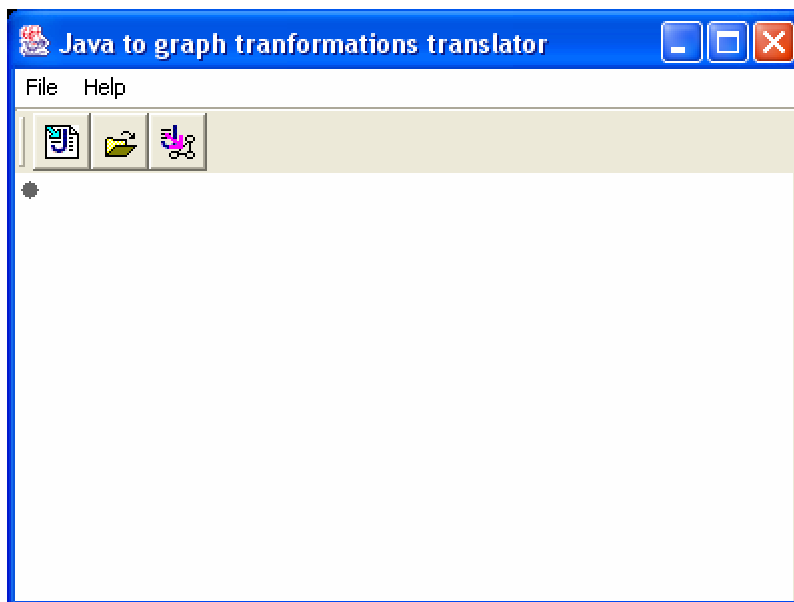
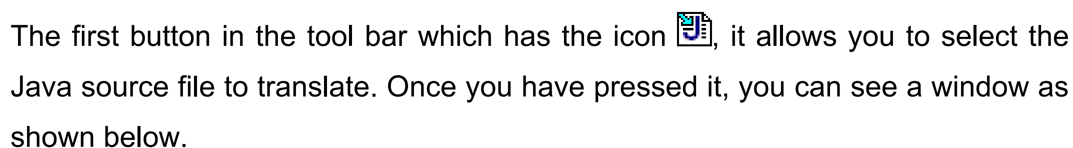


Figure 3.1.3.1. Starting the application

The first button in the tool bar which has the icon , it allows you to select the Java source file to translate. Once you have pressed it, you can see a window as shown below.

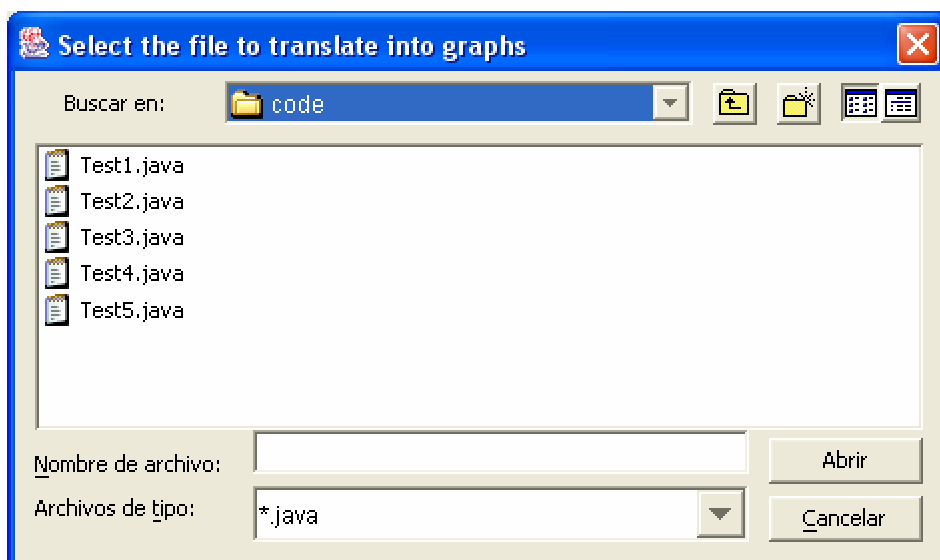
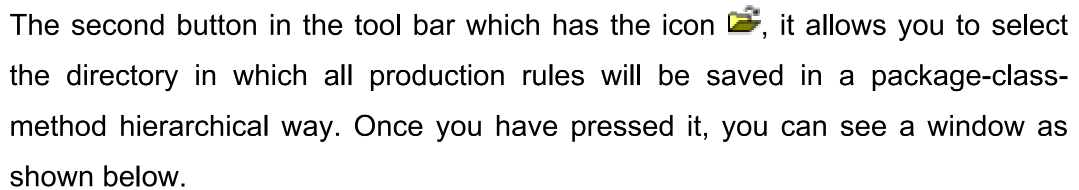


Figure 3.1.3.2. Loading Java file to translate

This window will show you only Java source files (*.Java) and directories. Once you have found the desired file to translate you can press the button labeled Open. If you click this option accidentally simply click the button labeled Cancel. It will not change the Java file to translate.



The second button in the tool bar which has the icon, it allows you to select the directory in which all production rules will be saved in a package-class-method hierarchical way. Once you have pressed it, you can see a window as shown below.

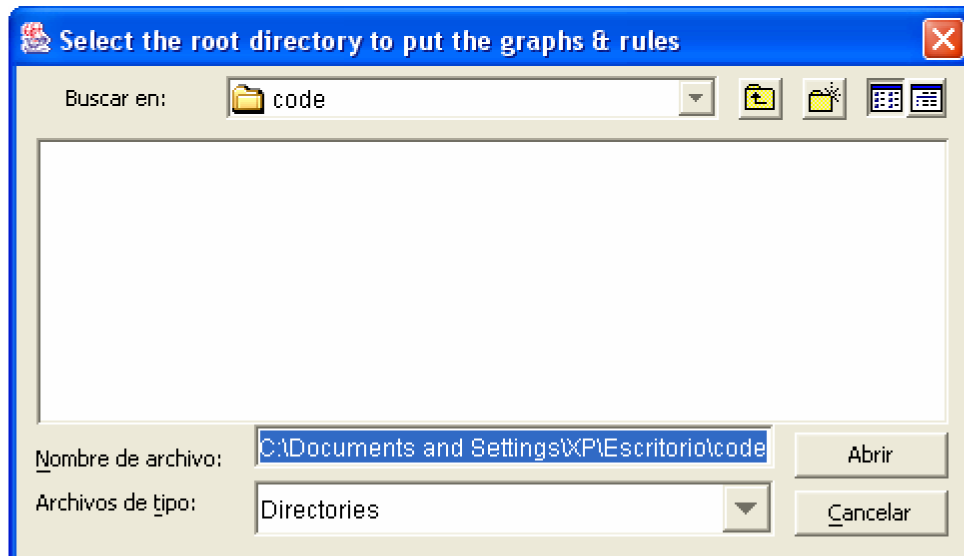
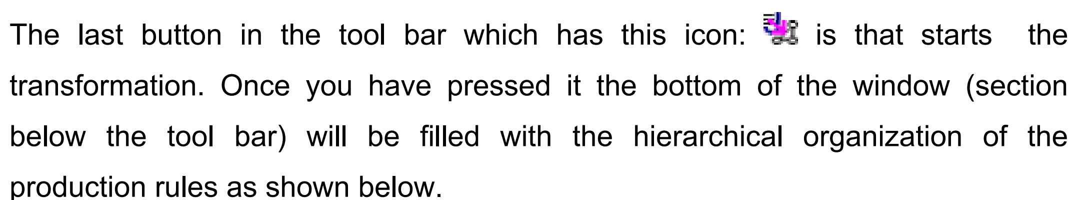


Figure 3.1.3.3. Specifying the desired target directory

This window will show you only directories. Once you have found the desired directory in which you want to save the production rules you can press the button labeled Open. If you click this option accidentally simply click the button labeled Cancel. It will not change the target directory.



The last button in the tool bar which has this icon: is that starts the transformation. Once you have pressed it the bottom of the window (section below the tool bar) will be filled with the hierarchical organization of the production rules as shown below.

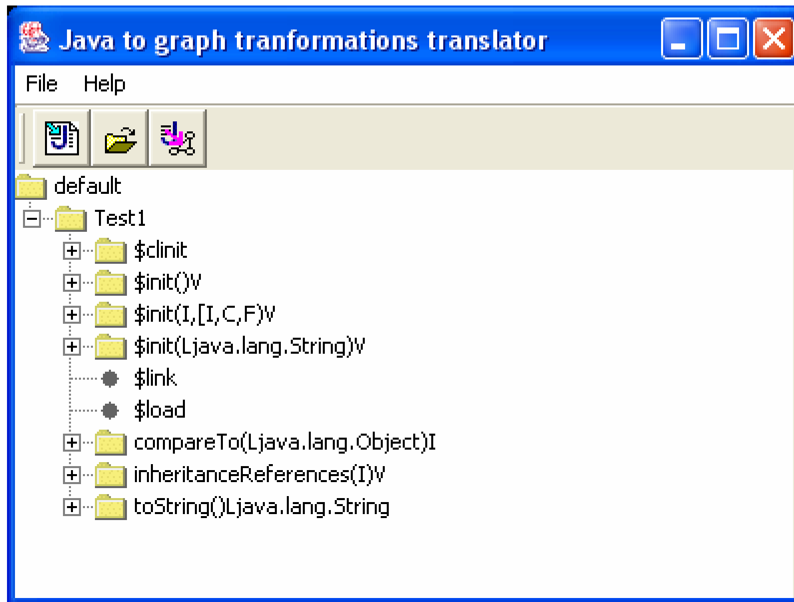


Figure 3.1.3.4. Translator output

If you have already translated a Java file, and you want to translate other Java files, you can repeat the process. In any time you can also change the desired target directory.

3.1.4. Output directory structure

The packages, classes and methods are represented as directories. The methods are subdirectories of the classes and the classes are subdirectories of their package directory. The packages also can be nested into other packages. In this way, the production rules' file structure also represents the ownership of a method to its class, the property of a class to its package and of a package to its parent package.

For example, let's consider:

```
package myPackage.mySubPackage;
import myPackage.MyInterface;

public MyClass
extends MySuperClass
implements MyInterface{
     String variable1;

     public void myMethod (String parameter1){
```

```
            variable1 = parameter1;
        }
    }
```

This Java source code would give as output the following directory structure:

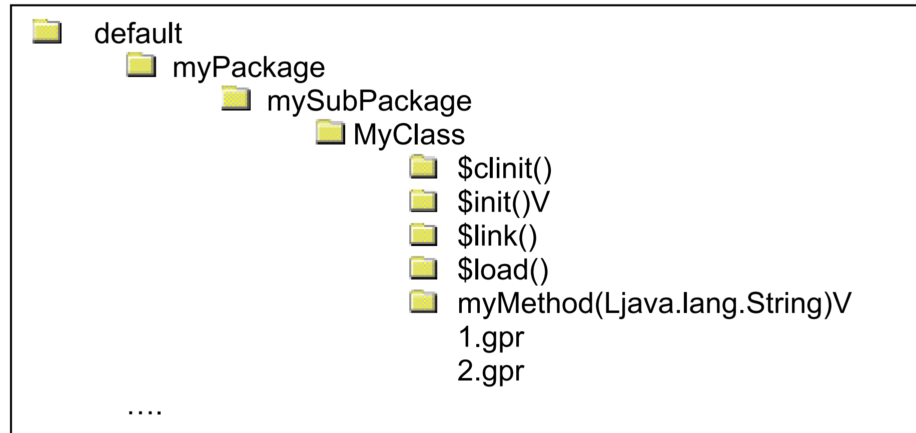


Figure 3.1.4.1. Translator directory structure

3.1.5. Reporting bugs

If you find any kind of error (bugs), you can report to groove_software@yahoo.com . In this way, you can help us to improvement the on coming versions of this application.

Fill out the following questions completely. The information requested is essential to solve the problem. Nevertheless, sending the error report is not a guarantee that the problem will be resolved.

Type of bug

Select:

- **Bug** if some feature of the product does not perform to the documentation.
- **Request** for enhancement if there is some feature not present in the product which you feel should be included.

Product:

Select:

- **Editor** if you have troubles editing graphs and graph production rules.
- **Viewer** if you have troubles viewing existing graph production systems, i.e., collections of graph production rules.

- **Simulator** if you have troubles simulating graph production systems, starting in a given initial graph.
- **Translator** if you have troubles mapping form Java source code to production rules.

Synopsis:

Enter a one line summary of your report. Please be specific.

Description:

Enter a detailed description of the problem. Please describe only one problem per report. For multiple problems, file a separate report for each one.

Frequency:

Select, how often does the bug occur?

- **Always**
- **Often**
- **Occasionally**
- **Rarely**

Steps to Reproduce:

Describe the step-by-step process we can follow to reproduce this bug.

Expected Result:

Describe the results you were expecting when performing the above steps.

Actual Result:

Please report the actual results that you saw.

Error Message(s):

Exact text of any error message(s) that appeared or any trace information available.

Severity:

Select, what impact does this issue have on developing your software?

- It is **impossible to continue** working without resolving this error.
- It is **difficult to continue** working without resolving this error.
- It is **possible to continue** working without resolving this error.
- **No Impact**.

User Info

Please give us some information about yourself. Be sure to include a valid email address. We will use this data for communications with you to clarify issues regarding the report you submitted and/or status of that report.

- **Name:**
- **Email:**

3.2. Programmer guide

Translator is a subsystem inside GROOVE tool. It translates Java source code to production rules.

Translator creates a hierarchical directory structure that represents the Java hierarchy: packages-classes-methods. The application save the production rule files inside the method directories to which they belong. These production files are XML files produced by GROOVE.

This section explains in general terms how the tool was built.

3.2.1. Requirements

Translator must create production rules that represent a Java program given its source code. These rules must also be compatible with GROOVE output format.

Performance is not a requirement for the first version of Translator.

3.2.1.1. Project risks

There is a technical risk because the main programmer requires significant learning of the theoretical foundation of the project as well as experience with GROOVE and Recoder.

Since the there are two persons working in order to achieve a similar kind of translation (Java source code and Java byte code to production rules) the communication needs to be carefully coordinated, efficient and effective.

3.2.2. Background

This section describes the software environment in which the Translator was developed that is to say GROOVE and Recoder architectures.

3.2.2.1. GROOVE

GROOVE follows a design style that contributes to its quality because it assures a strong cohesion and low coupling.
The high cohesion is achieved by having classes and methods with a specific well defined task; in this way following the application logic is painless. In GROOVE this characteristic is made evident by the class hierarchy; all concepts mapped by a class are first abstracted as their common interface, then if there are some common behavior features, they are encapsulated in an abstract class and, finally, the different kinds of a concept are represented as various classes that inherit the concept interface and concept abstract class. Additionally, each concept has a default representation. For instance, see Label (interface), AbstractLabel (abstract implementation) and DefaultLabel (default representation), in GROOVE's class diagram figure 3.2.2.1.

The light coupling is reached by reducing the interactions between a method and other objects, in other words, by giving each method a simple task it is an *accesor* or it is a *modifier* method. GROOVE provides low coupling by sharing utility functions that do not modify the parameters received, also by creating a different class for each kind of concept because it allows having custom-made methods.

Moreover, its documentation is complete in the sense that it describes the purpose of the code, its required and provided information and the required conditions to execute it. In this way, GROOVE pretends to maximize its reusability and extendibility

GROOVE is divided into 8 packages. Most of them are intended to model a conceptual layer in the system. For example: `groove.graph`, `groove.trans`, `groove.lts` and `groove.Translator`. The rest of them (i.e. `groove.gui`, `groove.io` `groove.jgraph` and `groove.util`) represent functionality layers provided as interface or convenience utilities for the conceptual packages.

The `groove.graph` package defines the basic concepts and their properties like graph, nodes, edges, labels. It also defines the concept of morphism.

The `groove.trans` package defines the production rule terms like the production rule and the negative application conditions, it does not define the created, required and deleted element concepts because this is set for default when a morphism is established. It also characterizes the derivation or rule application notion.

The `groove.lts` package is dedicated to define the labeled transition systems that are all possible derivations given a set of production rules.

The `groove.gui` package is responsible for the visualizing functionality; `groove.io` package is in charge of GROOVE's input and output functions; `groove.jgraph` package was developed to extend the graphs visualization and `groove.util` package is for general convenience functionality.

The Translator package contains the one developed in this project and is treated in the section 3.2.3.

The figure 3.2.2.1 depicts GROOVE's UML class diagram.

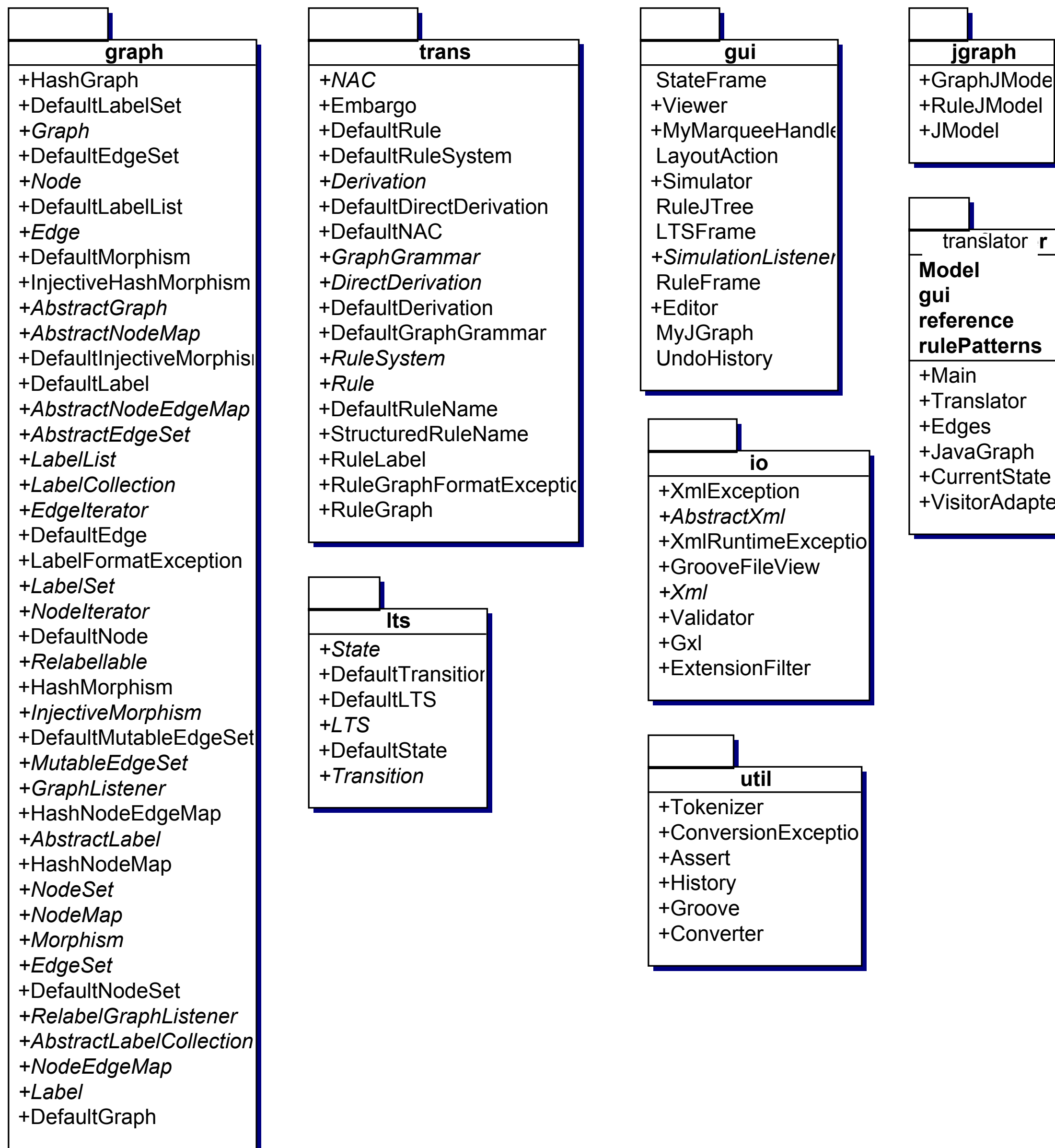


Figure 3.2.2.1. GROOVE's class diagram

3.2.2.2. Recoder

Recoder is built with a layer architecture, it contains the *Intermediate Representation Layer* that maintains the intermediate representation of the programs being manipulated, i.e., the model of the programs, on top of it the Refactoring Command Layer that maintains a worklist of the refactoring commands of the user, and provides a simple transaction concept for interactive use of Recoder.

The intermediate representation layer is conformed by:

Reading/Writing Layer: This layer reads and writes programs, classes from and to files. These are the layer components:

- Parser. The Recoder parser is generated from a JavaCC grammar.
- Java program factory. The parser calls factory functions of the Recoder abstract syntax tree (class Recoder.Java.JavaProgramFactory). Internally, the factory calls the constructors of the AST classes, but hides the actual form of the AST so that the AST can be easily exchanged.
- Pretty printer The pretty printer prints an AST to file or a stream. It is hand-written, and exists in several versions: Java pretty printer, html writer, JXML writer.

Source File Repository Layer:

The source file repository maintains all ASTs, including a mapping from file names (source file names, byte code file names) to ASTs. It starts the parser and the pretty printer if desired.

Definition objects handling

This layer handles definitions of the program (the abstract model). Definitions are types (classes and interfaces), methods, attributes, packages, and other semantic objects of Java programs. All these objects have been checked by the Recoder semantic analysis, i.e., are valid in terms of Java semantics. The main facade classes of this layer are

- Recoder.service.NameInfo: maintains the meaning of names, and can load classes from file, given their name
- Recoder.service.SourceInfo: maintains the relation between AST elements and semantic (abstract) objects, i.e., between source and abstract model.

Cross-reference info handling.

The cross reference info (class CrossReferenceService) links all definitions (variables, parameters, fields) to their references. For refactoring, this information is indispensable, to rename methods, fields, and classes.

The core of Recoder is its program model. It is structured in two packages: the Recoder.Java.* that contains the abstract syntax trees (AST) classes i.e. it

contains the source model and the Recoder.abstraction.* has the program definitions or abstract model.

3.2.3. Translator architecture

This section explains the architecture of Translator. Each section describes the purpose of `groove.Translator` package and how it is reached.

The `groove.Translator` package encapsulates the code transformation into production rules. This package is in charge of direct the transformation process.

This process starts when a frontage class (`groove.Translator.gui.Application1`) calls the main method in the Main class, this method creates a Recoder.ServiceConfiguration object and set its properties, then it calls a method that given a Java source file, it returns the AST root node.

A new Translator object is created with the AST root node, the output directory and the service configuration. It has a field of type CurrentState inherited from its super class. When the Translator object is created calls a method in the Current State that performs a preprocessing transformation that gives as result a class graph of the Java source code.

Then, the main method starts the AST tree visitation.

The AST navigation and visiting is performed by the Translator and its super class. While the tree is traversed the production rules are generated. Depending on the kind of node to which the visitor arrives; if the node is one of the modeled instructions, the production rule is created and it is applied to an object graph (which represents an instance of the Java source code that is been processed). This object graph is accessed through the current state field. If the node is not modeled the application will anyway visit its children and printing in console that it visited those nodes.

Each time that a production rule is generated it is saved to inside its corresponding method directory, and its name is the instruction edge label.

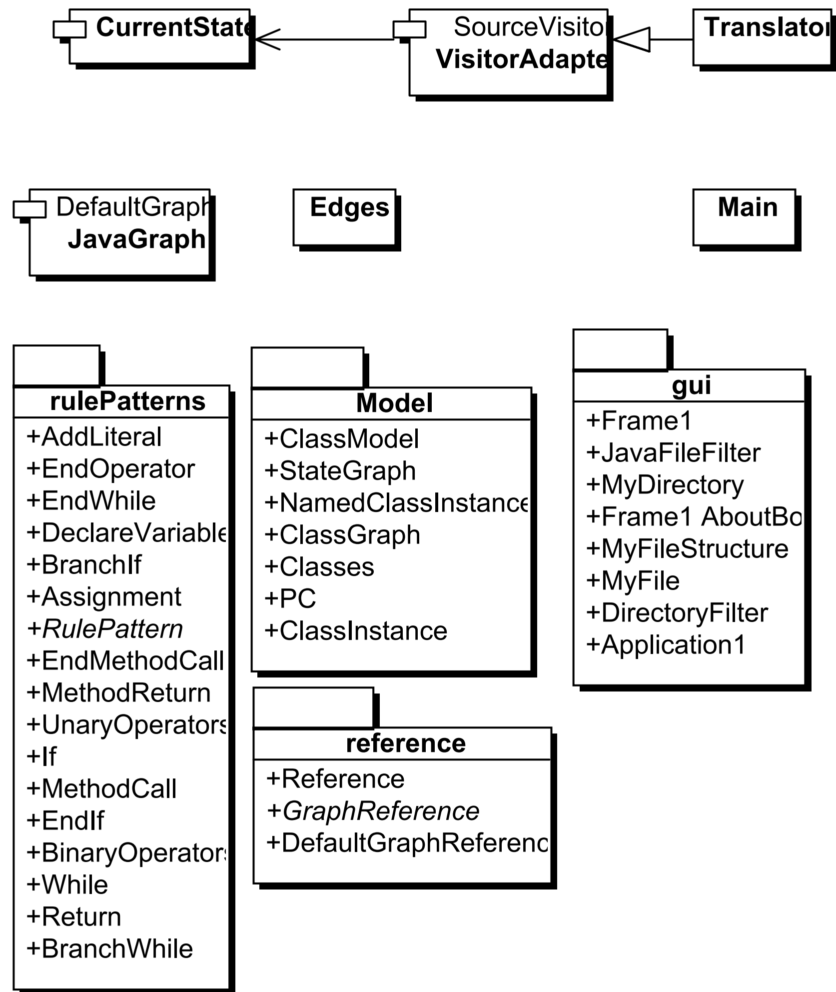


Figure 3.2.3.1. Translator package class diagram

The diagram 3.2.3.1 shows the Translator sub packages and its classes.
The Edges class stores in constant fields (abstract final) the name of all edges defined to represent Java code (for example instanceOf, super, PC, active, class, etc). The JavaGraph class of DefaultGraph class, it has some methods that allow asking for graph elements as their 'name' or their relationships with other graph elements.

3.2.4. Subsystem Design

This section states how the packages and classes contribute to the translation process in order to get a clear idea of each package purpose.

3.2.4.1. Model package

The model package (`groove.Translator.Model`) is used to take advantage of the Java model information provided by Recoder in order to keep all

the required information about the class that is being processed and transform is into a class graph. It also provides a class that represents the object graph that is being translated to maintain its information in a consistent way.

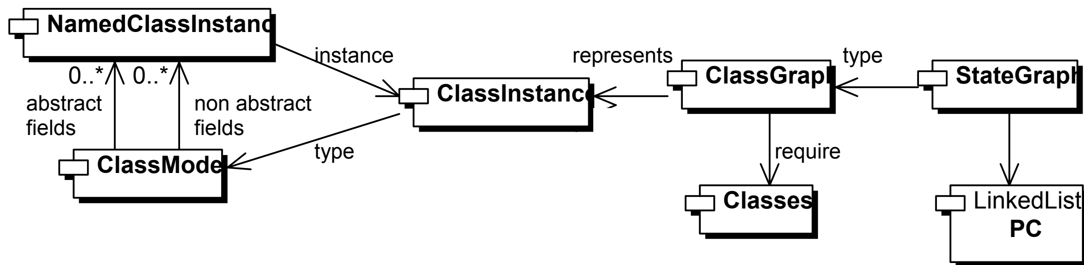


Figure 3.2.4.1.1. Model package class diagram

ClassIntance, ClassModel, NamedClassInstance and Classes types model the processed Java source into a class graph into the ClassGraph class, with this graph the StateGraph class creates a new object graph that is the one that keeps the necessary information to generate the production rules. PC class models the sub graph that carries the instruction ordering information.

3.2.4.2. Reference package

Reference package (`groove.Translator.reference`) is in charge of pointing to a graph element in or outside a context.

Given that a graph element has no identity itself to be able to mention a graph element, for example inside a rule application, is necessary to create another object that points to the desired graph element reference. In some cases the graph element require some additional information to ensure that it is the desired one, for example a method instance can be identified unambiguously by its method node, its defining class node and its caller object. These graph elements are collected in a sub graph that creates this differentiation pattern, among these graph elements build a context in which the desired element is the one pointed by the reference.

A context-less reference is represented by the `Reference` class and a context reference is represented by the `DefaultGraphReference` class. The context reference interface is defined in the `GraphReference` class. The corresponding class diagram is depicted 3.2.4.2.1.

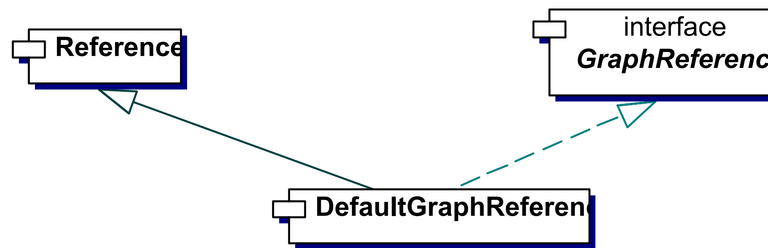


Figure 3.2.4.2.1. Reference package class diagram

The references are required because the Abstract Syntax Tree realizes if a software entity was previously defined, in such cases the visited node will not have the element declaration information but a reference to its name and in some cases other relevant information like its type name or its invoker name. W e call ***invoker*** to the caller object, in other word is the one that refers its messages or fields by the dot (".") syntax construction, for instance, object.myField or object.myMethod(). This node is the one that permit the unambiguously location of a reference.

If there is not any explicit invoker, it is assumed that the default one is the *this* reference or object instance of the processed Java code.

The following sections will explain briefly each type of referenced refer in a Recoder Abstract Syntax Tree and how it is represented unequivocally.

3.2.4.2.1. Field

A field reference (`object.fieldName`, in source code) can be easily recognized once the object to which belongs and its name are known, as shows the following illustration. The referenced element is the one with a narrow line.

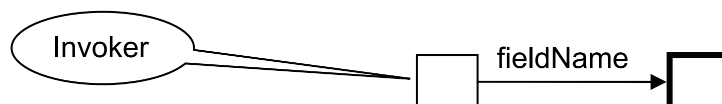


Figure 3.2.4.2.1.1. Field reference pattern

3.2.4.2.2. Class

A class reference (`ClassName.class`, in source code) is identifiable with its name. This is illustrated in the following graphic.

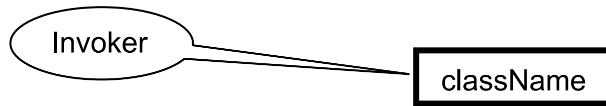


Figure 3.2.4.2.2.1. Class reference pattern

3.2.4.2.3. *Method*

A method reference (`object.methodName(parameters)`, in source code) is differentiated by its signature, caller object and its defining class. The picture 3.2.4.2.3.1 depicts a method reference, note that the reference is the narrowed element.

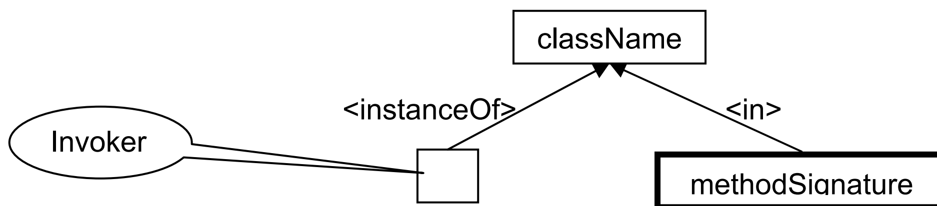


Figure 3.2.4.2.3.1. Method reference pattern

3.2.4.2.4. *Super constructor*

A super constructor reference (`super()`, in source code) is acknowledged with its signature and the caller object. This can be observed in the following diagram.

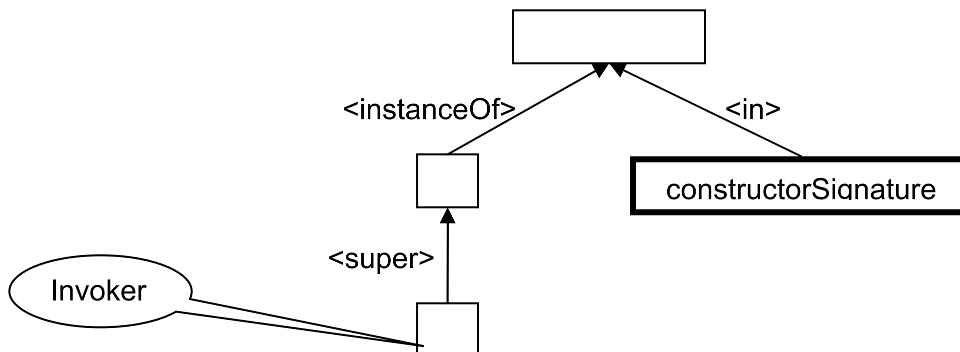


Figure 3.2.4.2.4.1. Super constructor reference pattern

3.2.4.2.5. *Super*

A super reference (**`super.`**`something`, in source code) is recognized with the caller object as shows the figure below.

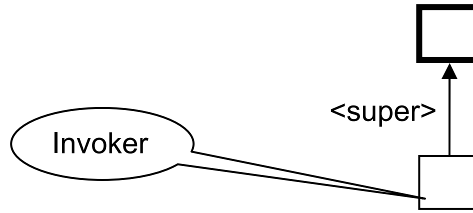


Figure 3.2.4.2.5.1. Super constructor reference pattern

3.2.4.2.6. This constructor

A this constructor reference (`this()`, in source code) is identifiable with its signature and the caller object, as obvious this is a mix between the this and the method reference patterns, it can be verified in the next picture.

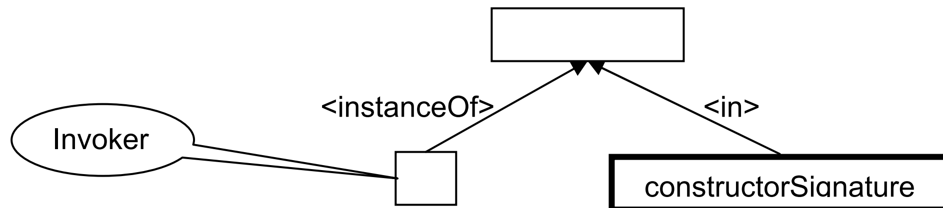


Figure 3.2.4.2.6.1. This constructor reference pattern

3.2.4.2.7. This

The this reference (**`this.`**`something`, in source code) is the same invoker or caller object.

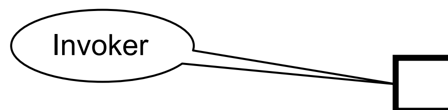


Figure 3.2.4.2.7.1. This reference pattern

3.2.4.2.8. Type

A type reference (`ClassName`, in source code) is differentiated by its name, because the class nodes are named with fully qualified names, as the next illustration shows.

Figure 3.2.4.2.8.1. Type reference pattern

3.2.4.2.9. Variable

A variable reference (`variableName`, in source code) can be easily acknowledged once the object to which belongs and its name are known, as shows the following image. The referenced element is the one with a narrow line.

Figure 3.2.4.2.9.1. Variable reference pattern

3.2.4.3. Rule patterns package

The rule patterns package (`groove.Translator.rulePattern`) is is the one that effectively converts an object graph and some references to it into a production rule file, actualizing the information inside the object graph in order to be consistent to the next instruction, i.e. the object graph has applied the instruction transformations. This is accomplished by using a class that contain all the general functionality of a production rule pattern like saving the rule into a GROOVE file and a class for each specific transformation rule pattern that creates the production rule and applies the pattern to the object graph.

To know more about the production rule patterns see section 2.4 that explains each instruction and its translation into a production rule pattern.

The class diagram below presents the implemented rule patterns in Translator.

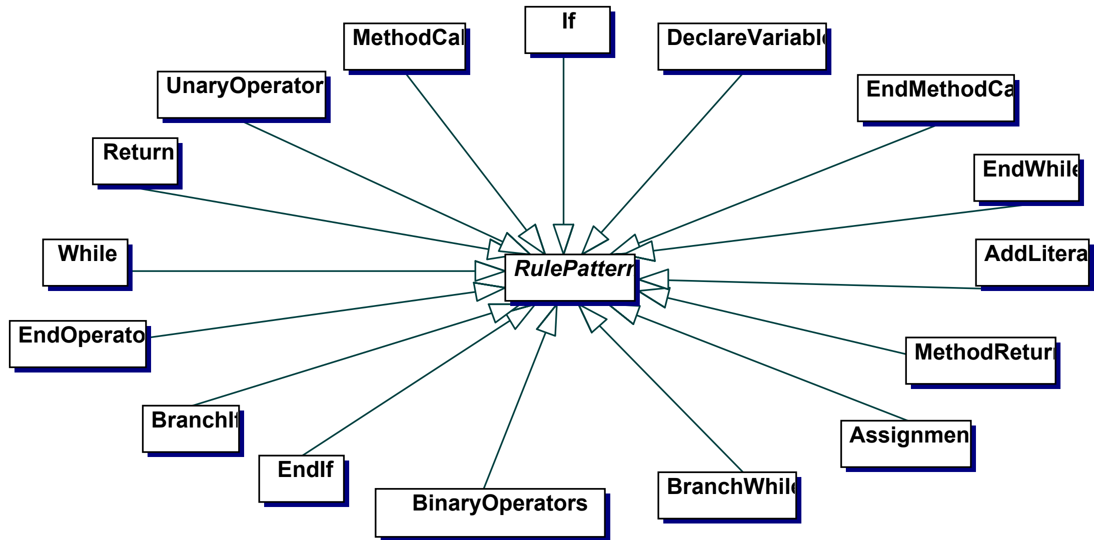


Figure 3.2.4.3.1. Reference package class diagram

3.2.4.4. GUI package

This package (`groove.Translator.gui`) contains the set of classes created to generate the application interface. The following figure is the gui package's class diagram Figure 3.2.4.4.1.

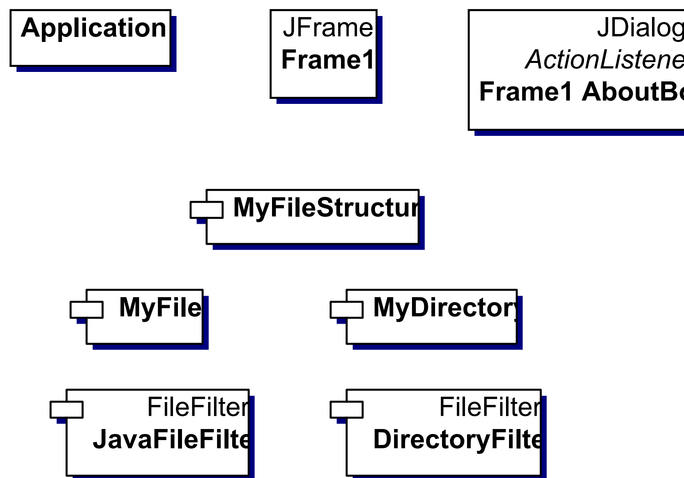


Figure 3.2.4.4.1. Reference package class diagram

The main class in this package is Application1 which is responsible of performing the input / output functionality by collaborating with the other classes in the package.

The Frame1 class is the output frame in which the generated code tree is inserted; the Frame1AboutBox is a window with some information about Translator. The FileStructure, MyFile and MyDirectory are convenience classes to show in a tree way the output files, and finally, the JavaFileFilter and DirectoryFilter are classes that permit showing or not Java source files and/or directories depending if the user is selecting the file to transform or the output directory.

3.2.5. Getting involved in the project

The following section describes the procedure to follow in order to get access to the Translator source code, and what changes you have to make to your environment before fetching the sources from CVS. It also establishes a few basic rules you have to follow, when updating CVS with modified versions of Translator source files.

3.2.5.1. How to join the project

This project is promoted by Arend Rensink associated professor of the computer science department at Twente University.
Actually there is another parallel project being developed by Mark Arends (m.r.arends@student.utwente.nl) whose purpose is to translate Java byte code into graph transformations.
Your comments and suggestions are well received.

3.2.5.1.1. Last CVS version

GROOVE source code (including Translator) is maintained on a machine named demeter.cs.utwente.nl using CVS. This machine is from Twente University, the GROOVE directory and CVS administrator is Arend Rensink.
Follow these steps to get involved:

1. Send an email to Arend Rensink (rensink@cs.utwente.nl) asking for an account on demeter.
2. Once you have obtained the account, you need to set up the following environment variables to get ready for a CVS checkout

   CVSROOT
   :extssh2:account@demeter.cs.utwente.nl:/home/trese/projects/GROOVE/cvs
   CVS_RSH ssh2
3. Go to your home directory and type:

   cvs checkout src

This will checkout all sources, binaries and other files into a directory tree whose root is home/GROOVE/src

4. If you want a particular version proceed as above, but use command:

   cvs checkout -r versionid src

   Where, versionid is the name of the version you need. (Ask Arend Rensink for available versions).

5. In order to commit changes back to CVS, you need to do the following:
   - Make sure you tested your changes before
   - If you do not have your own branch and you made substantial changes, you should show your results first to Arend Rensink.
   - Always add a comment stating the changes you made, when cvs is asking for it (after a cvs commit). This will allow identifying, who changed what and for what reason.

3.2.5.1.2. Directory structure

This section it says where is the information that concerns to this project.

Directory	Content
/src/GROOVE/	This is the directory where the GROOVE source files starts.
/src/GROOVE/Translator	In this directory are the Translator source file classes and sub packages directories
/src/GROOVE/Translator/gui	This directory contains the classes created for the user interface in Translator.
/src/GROOVE/Translator/reference	This directory has the classes that point to a graph element or set of elements.
/src/GROOVE/Translator/model	The model directory holds the classes in charge of representing a Java class in a textual way and converting it into a class graph.
/src/GROOVE/Translator/rulePatterns	Inside the rulePatterns directory is stored the most important part of the project : the generic production rules that given an object graph and some references inside it, produce the production rule file and

	the next object graph (after applying the instruction transformation)
/src/GROOVE/Translator/tests	The tests directory contains the unit tests created in order to test the Translator functionality.
/src/classes/	This directory has a structure similar to the Translator directory, but it does not contain Java source file but its corresponding binaries.
/src/resources	In this directory is saved all files required to execute any of the GROOVE tools, for example the icons.
/doc/	This directory possesses the code documentation.

Table 1.2.5.1.2.1. Translator's directory structure

3.2.5.2. How to extend the project

There are two possible extensions for this project the first one is to model other Java statements as production rules; this will have as consequence the addition of new production rules. The other one is to change the parser (now the parsing is performed with Recoder) to another more powerful or with a closer representation to the Java source model. This section analyzes the necessary modifications in order to extend them.

3.2.5.2.1. Adding new production rules

To add a new production rule is necessary to create the class that converts the rule pattern in specific production rules for a given object graph.
It is also necessary to check in which node of the AST must this new rule be called, being carefully to check if it has all necessary references.

3.2.5.2.2. Changing parser

Right now Translator is strongly attached to Recoder parser, to be able to change it is mandatory to create a wrapper class that converts from the parser AST output to a more generic AST.

4. Processing algorithms

In order to obtain an Abstract Syntax Tree that organize and interpret the java source file, Translator uses Recoder. Translator also uses GROOVE graphs and production rules model to represent its own data. For these reasons, Translator is in charge of three main tasks: configure of Recoder to parse the java source code and obtain the Abstract Syntax Tree (AST),; preprocess the AST to get the java class in a graph representation, and generate the production rules by navigating the AST.

4.1. Parsing and Abstract Syntax Tree Construction

This section offers an explanation of the process that Recoder executes in order to translate a Java source code into it corresponding Abstract Syntax Tree mapping. See [5]

First, the lexical analyzer takes successor character relation and groups several characters together to tokens.

Then, parser analyses the successor token relation according to a context free grammar and produces an abstract syntax tree. Firstly, some of the tokens are skipped (such as keywords or comments) and others are retained as new objects of the new graph (for example `if, while`, procedure). Between those, a new tree-like relation is formed, the abstract syntax graph that contains: definitions of objects, references to objects, statements which compute something, expressions which are side effect free statements, and groupings which group definitions, references, statements, and expressions to larger blocks.

After that, on the abstract syntax tree, a semantic analysis process starts. This process relates some of the elements of the tree to others which are remote (this is, elements that have been declared in other place of the tree). Hence, the abstract syntax tree is generalized to a graph. Semantic analysis constructs several graphs which build on top of each other:

- Object oriented languages define an inheritance relation which records reuse references between classes and defines an extended visibility concept for class features (all features of the super classes are visible too).

- Import relation (visibility relation). Beyond the module that is being translated, other objects in other modules may be referenced, if they are imported. The front end computes a relation along which names are looked up.
- The *use-def* graph of names. For every name used in the program (identifiers or structured identifiers called *designators*, also called *references*) the application should check whether it is defined once, which type it has and whether it shadows other definitions of the same name. This process usually is called name analysis, and it builds up a relation between all used names and their definitions. Conceptually, inheritance and import relations can be considered as parts of the use-def information, however, for practical reasons they are separated.
- With the use-def graph, expressions and statements, procedure calls and generic parameterizations can be type-checked: just compare the types of the formal to the types of the actual arguments (type checking).

4.2. Preprocessing

The preprocessing stage takes the Recoder meta programming information in order to generate a class graph of the processed source file that later on serves as base for generating a new object graph in which the production rules that represents the instructions in its respective source code can be applied.

The preprocessing maintains in the `groove.Translator.Model` classes all the elements that compose the processed class and its required classes like: super class, interfaces, field types, parameter and return types, etc. For this purpose it uses the meta information provided by Recoder that allows operations like `compilationUnit.getDeclarations()`, `compilationUnit.getImports()`, `classType.getFullName()`, `cd.getAllSupertypes()`, `classType.getConstructors()`, `classType.getMethods()`, `method.getSignature()`, `method.getReturnType()`, etc.

All class information is used to generate the directory structure (see section 3.1.4) required to save the production rule files and it is saved in an instance of the ClassModel. All processed classes are in a dictionary whose key is the fully qualified name of the class and whose value is its corresponding ClassModel instance. This dictionary is the Classes class.

If along the processing of any class is found a new required class it is processed and saved in the dictionary. This approach can be optimized because although it does not allow circular reference looping it could overflow the call stack.

Once all required classes information is extracted, the class graph is generated. This process is performed by asking the ClassModel instance of the class that is being translated to produce its graph representation. A ClassModel graph representation is a graph reference (see previous chapter programmers guide section 3.2.3) whose referred element is the class node. When an element of the class that is being processed is going to be added, it calls the generate graph representation method of its corresponding ClassModel, previously stored in the dictionary. In this way the graph representation call is propagated through all required classes that are added as part of graph representation of the element.

4.3. Production rules generation

Production rules generation is carried out as follows:

1. A new production rule is declared with its required references and the actual object graph.
    a. The object graph is cloned into a field called clone
    b. The clone is cleaned form the non required literals in the rule
2. The apply transformation rule is called
    a. A graph representing the instruction order is added to the clone
    b. A new local variable called lhs, which represents the left hand side of the graph transformation, is created from the clone
    c. A new local variable called rhs, which represents the right hand side of the graph transformation, is created as clone of the lhs
    d. A new injective morphism is established between the lhs and the rhs
    e. If the references are complete
        i. The production rule changes are applied to the rhs.
        ii. A new production rule is created from the morphism
        iii. The production rule is saved. This is accomplished with the GROOVE class Gxl that converts a GROOVE graph into its corresponding GROOVE XML.
        iv. The production rule changes are applied to the actual object graph.
    f. If the references are not complete

i. Only the production rule changes that modify the instruction order are applied to the rhs. This is done in order to maintain the instruction order consistency.
ii. A new production rule is created from the morphism
iii. The production rule is saved. This is accomplished with the GROOVE class Gxl that converts a grove graph into its corresponding GROOVE XML.
iv. Only the production rule changes that modify the instruction order are applied to the actual object graph.

This application of the production rules generated to the object graph is fundamental because it allows an object graph with the expected object state needed for the rule matching, in other words it helps to keep the object status required for applying certain rule, for example if a method defines a local variable, the processing algorithm must know that this variable is only accessible inside the method scope, and which is its type and value as well as the objects that refer to it.

5. Future work proposals

This section point's out areas of future work. It first presents a detailed list of Java language features that are not currently implemented. Next, some areas in which possible refactorings were identified. These refactorings are focused on producing more extensible and reusable code and some improvements concerning the process report. A last part, suggests some priority on the missing tasks.

5.1. Missing or incomplete features

This section identifies those parts of the Java language that are not currently modeled or available in Translator.

Translator omits access modifiers that could be present in the source program by setting default access modifiers instead. The reason for this is that the model presented is not complete to model the modifiers semantics.

Feature	Comments / JLS Reference

New operator	It allows the translation of class instantiation
	`new Object ();`
	`JLS reference: 15.9`

Load	It simulates the search of binaries done by the virtual machine JLS reference: 12.2
Link	It helps to map the process of verifying class loading, preparing memory for the execution and resolving references to other classes. JLS reference: 12.3
Abrupt completion	Permits the modeling of errors during expression evaluation and statement execution. JLS reference: spread in 4.
Modifiers	Models the accessibility of members, it changes the semantic and representation of an object. For example abstract fields must belong to their class node, while non abstract ones must belong to their instance node. **`abstract; final; native; private; protected; public; static; strictfp; synchronized;`**

	transient; volatile JLS reference: 8.1.1 - 8.3.1 - 8.4.3 - 8.5.1 and 9.1.1
Arrays	Their representation helps to have collection of fixed size with object of the same type. JLS reference: 10.
Threads	With them is possible to model concurrency. JLS reference: 17.
Missing statements	Execution flow control Labeled: **break**,**continue** JLS reference: 14.14 and 14.15 respectively. Jump: **switch -case** and **default-**; **do**; **for** JLS reference: 14.10 – 14.12 and 14.13 respectively. Exception: **throw, try -catch** and **finally-** JLS reference: 14.17 and 14.19 respectively. Concurrency: **synchronized** JLS reference: 14.18. Expressions: **post increment, post decrement, pre increment, pre increment.** JLS reference: 14.8.
Inner classes	Representing inner classes. JLS reference: 8.1.2.

Table 5.1.1. Topics for completing java-graph model

The table 5.1.1 presents the java features that are not present in the model. The purpose of remarking these points is to provide some directions in which this work can be extended.

There are some topics that were discussed and slightly modeled during this work, such as link, load, abrupt execution and garbage collection, but given the lack of time it was not possible to achieve a stable representation, for that reason they are not part of the document neither of the tool.

5.2. Improvements

The following improvements should be considered or need to be done, concerning the Translator architecture.

Given that GROOVE is an extensible project, each one of its components should follow GROOVE's design policy. The purpose of this section is to state some

important steps previous to any further extension, they are simple arrangements of the existing code that would give a more reusable code.

Artifact	Comments	Benefit
Translator	It is necessary to separate it into two packages one in charge of creating, initializing and wrapping the parser and another one in charge of traversing and processing the AST.	Less coupling
Translator.gui	Some functionality in this package was already implemented in groove.gui, therefore it is necessary to move the non implemented classes and to eliminate the replicated code.	Extensibility
Translator.model	This package also needs to be divided into two smaller packages: one for a more detailed java semantic modeling (ClassModel, ClassInstance, Classes, etc.) and another for having a more intuitive and clean representation of java graphs (JavaGraph, Edges).	Higher cohesion
Translator.references	GraphReferences class must be modified in order to eliminate duplicated information as main and name, whose equivalent is defined in Reference class	Extensibility

Table 5.2.1. Refactoring improvements

The current process report needs to be improved. Right now, only shows the final file structure produced by Translator nevertheless it would be better to show which line of code is processing and how is going to translate it.

5.3. Priorities

In order to aide in the future development of the tool it is compulsory to evaluate which of the tasks to perform must be done first. The purpose of this section is to evaluate which of the mentioned additions or improvements contribute more to extend the scope of programs that can be processed with Translator.

Priority	Tasks	Type	Importance	Difficulty
1	New operator	Java modeling	High	Low
2	Link	Java modeling	High	Medium
3	Modifiers	Java modeling	High	High
4	Load	Java modeling	Medium	Medium
5	Abrupt completion	Java modeling	Medium	Medium
6	Missing statements	Java modeling	Medium	Medium
7	Translator.model	Code refactoring	Medium	High
8	Translator	Code refactoring	Medium	High
9	Translator.gui	Code refactoring	Low	Low
10	Translator.references	Code refactoring	Low	Low
11	Arrays	Java modeling	Low	Medium
12	Inner classes	Java modeling	Low	Medium
13	Threads	Java modeling	Low	High

Table 5.3.1. Priorities of tasks to do.

6. Conclusions

As was established in the chapter 2, Java source code can be transformed into flat graphs; furthermore, this graph representation allows the inclusion of runtime information, which is not present in other representations such as Abstract Syntax Trees.

The produced model permits the simulation of code by instantiating rule patterns that represent Java statements transformations. This model: Java representation as graphs and production rules is unified with the byte code model. The source code and byte code models share the abstract Java graph model that serves as interface between both transformations.

Although the Java language specification is not fully mapped, the fundamental statements were constructed and their extension will not represent a big effort.

The Java source structures modeled were: declarations, types, fields, methods, parameters, method body, variables, expressions, literals, operators, references and some statements (if, while and other *expression statements* -assignment and method invocation-). In addition, the execution order was modeled.

There are some Java structures not represented in the model: modifiers, threads (synchronization), exceptions, inner classes and the rest of statements (*break*, *continue*, *do*, some *expression statements* –pre-increment, pre-decrement, post-increment, post-decrement, class-instance-creation-, *for*, *labeled statement*, *r*eturn, *switch*, *synchronized*, *throw*, *try*.).

Almost all modeled features, except the class loading and the implicit Java classes[9], were implemented in the tool. As a result, simple programs can be translated into production rules.

The production rule files generated by the tool developed in this project are compatible with GROOVE files, in this way; they can be seen, simulated and edited by the other tools that are part of GROOVE (Viewer, Simulator and Editor respectively).

This simulation (or successive application of production rules) shows the execution of a Java program by applying the generated production rules. This

[9] The implicit Java classes are those that can be referenced without import clause. For instance, java.lang.System

simulation maintains all the runtime information not present in most of the verification tools.

The main contribution of this thesis was to represent object-oriented programs (in particular Java source code programs) in graphs whose elements don't have own identity.

7. References

[1] H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner and A. Corradini. Algebraic approaches to graph transformation, part II: Single pushout approach and comparison with double pushout approach. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation*, volume I: Foundations, pages 247 – 312. World Scientific, Singapore, 1997.

[2] James Gosling, Bill Joy, Guy Steele, Gilad Bracha. The Java Language Specification, Second Edition, 2000. Available at http://java.sun.com/docs/books/jls/index.html

[3] A. Rensink. Model Checking Graph Grammars.

[4] A. Rensink. GRaphs for Object Oriented VErification: A tool set for the simulation and analysis of graph grammars. Available at http://www.cs.utwente.nl/~groove

[5] Recoder. Java framework for source code metaprogramming. Project URL: http://recoder.sourceforge.net/