University of Twente

Department of Computer Science

A simulation of the Java Virtual Machine using graph grammars

Master of Science thesis

M. R. Arends, November 2003

A simulation of the Java Virtual Machine using graph grammars

Master of Science thesis

M.R. Arends

Enschede
November 2003

Chair:	Software Engineering
Department:	Computer Science
University:	University of Twente
Graduation comittee:	prof. dr. ir. M. Aksit
	dr. P.M. van den Broek
	dr. ir. A. Rensink

Summary

Model checking is used to find problems in software. However the run-time behaviour of a program is poorly covered by existing model checking. To be able to analyse this run-time behaviour we want to be able to translate a Java program into graph grammars.

This report especially focuses on the translation of Java byte code into graph grammars. To be able to do this translation, a run-time state of a program has to be represented in a graph. For this representation a Meta model is developed.

A translator has been designed and implemented to create graphs and graph production rules corresponding to this Meta model. Using this translator simple Java programs can be translated to graph grammars. Currently not all JVM instructions and Java concepts are implemented; this is planned for the future. The templates used for the translation are very complex and need to be systematically designed in the future.

This translator can contribute to better software analysis and in the end help to find more bugs.

Contents

Summary **3**

1 Introduction **6**

2 Graph transformations **8**

2.1 Graphs 8

2.2 Graph production rules 8

2.3 GROOVE 10

3 Translating Java **14**

3.1 The Java Linked List example 15

3.2 The Java Virtual Machine 16

4 Representation **20**

4.1 Representation of different Java aspects 20

4.2 Meta Model 29

4.3 Loading, linking and initialisation 31

4.4 Method invocation and execution 34

4.5 Limitations 38

5 Implementation **40**

5.1 Design of the translator 40

5.2 Testing 46

5.3 Usage 49

5.4 Limitations 50

6 Conclusions and recommendations **51**

Bibliography **53**

A Linked list example source code **54**

B Linked list example byte code **57**

C Description of all Nodes and Edges **65**

C.1 JVM Node 65
C.2 Interface Node 66
C.3 Class Node 66
C.4 Object Node 67
C.5 Method Signature Node 68
C.6 Method Frame Node 68
C.7 Instruction Order Node 69
C.8 Stack Node 69

D Implemented JVM instructions **70**

D.1 Stack operations 70
D.2 Arithmetic operations 71
D.3 Control flow operations 72
D.4 Load and store operations 73
D.5 Field access operations 74
D.6 Method invocation operations 74
D.7 Object allocation operations 75
D.8 Conversion and type checking operations 75
D.9 Other operations 75
D.10 Reserved operations 76

E CD-ROM **77**

Chapter 1

Introduction

As software becomes larger and more complicated, there is ever more need for testing and analyzing the software. By testing and analyzing the software you can verify the software and find problems before it is used in a production environment. One way to analyze the software is model checking, in particular object oriented model checking.

Two examples of software model checkers are Bandera [1] and Java Pathfinder [9], both have been successfully applied at several research institutes. However, dynamic (de)allocation, due to object creation, garbage collection, method calls and returns is poorly covered by existing model checking.

In [11] it is proposed to use graph grammars to generate a transition system consisting of graphs as states and partial graph morphisms as transitions. This graph grammar consists of an initial graph and a set of production rules. This initial graph represents the first state of a program; the start of the execution. The production rules represent the different steps taken in the execution of a program. Control edges are used to ensure the production rules can only be applied in the right order. These control edges are comparable to the program counter of the Java Virtual Machine.

Using this graph grammar it is possible to create a model of all the run-time states of a program by generating all possible transitions from an initial state. This generated graph transition system is a simulation of how a program is executed in the Java Virtual Machine, so all the run-time states of a program can be analysed.

Before this graph transition system can be created, a program needs to be translated into a graph grammar, thus in an initial graph and graph production rules. To be able to do this, it must be known how represent a state of a program in a graph. This report will give a representation of the run-time state of a Java Program. By using this representation it is possible to translate a Java program into graph grammars.

This report focuses on how to translate Java byte code into graph grammars. The translation of Java source code is done in another project [12]. Also a translator for automatic translation of Java byte code into these graph grammars has been implemented. This translator is now part of the GRaphs for Object-Oriented VErification (GROOVE) project [10]. Most figures in this report are produced with the simulator that is also part of the GROOVE project.

First in Chapter 2 a general introduction to graphs and graph production rules is given. Also a description of the GROOVE toolset is given. Chapter 3 describes the Java class file format and

the Java Virtual Machine. Also the example used throughout this report is introduced. How to represent a program in a graph is described in Chapter 4. In Chapter 5 the implementation of the translator is given. Conclusions and recommendations are in Chapter 6.

Chapter 2

Graph transformations

2.1 Graphs

Graphs are mathematical models with a nice graphical representation. Graphs consists of nodes and edges, this can graphically be represented by boxes and arrows connecting them.[1] The nodes and edges can also have labels. An example of a graph is given in Figure 2.1. In this example there are three nodes, two with labels, and one without. Also two edges are in the graph, both with a label.

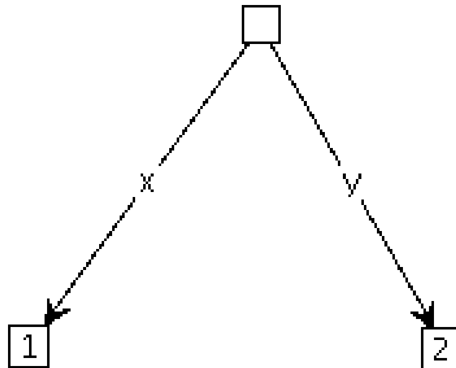

Figure 2.1: Example of of a graph

2.2 Graph production rules

When a graph is changed this is called a graph transformation. Every change in a graph will result in a new graph. An instance of a graph transformation establishes a relation between two graphs, the source graph and the target graph of the transformation. An example of a very simple graph transformation is shown in Figure 2.2. In this example one edge is deleted: the

[1] More accurately, this describes the special class of *directed graphs*, to which we limit ourselves here

edge with label *x*, and one new edge is created: a new edge named *x* from the node with no label to the node with label *2*. The label *y*|*x* is an abbreviation for two edges, one with label x and one with label *y*.

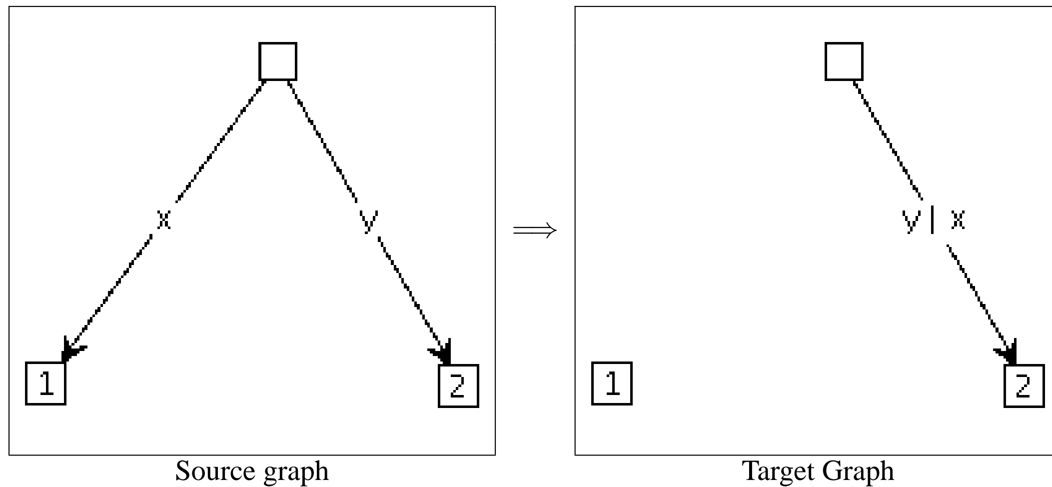

Figure 2.2: Example of a transformation instance

The example shows one instance of a single transformation, however normally we are interested in patterns of transformations that can be applied on many different source graphs and may result in may different target graphs. Such a pattern is called a *production rule*. A production rule describes how a graph is changed, but can be applicable to different source graphs. An example of a production rule for the above example can be: delete a edge with label x and add a new edge with label x to a node where an edge with label y is pointing. The only difference between the production rule and the instance given above is that we are not interested in the exact labels of the nodes. The production is given in Figure 2.3. Also the matching is depicted between the source and the target graph by dashed lines; this is done to make visible what the same nodes are in the source and the target graph.

There are different ways to represent and interpret production rules, however the following general principles for production rules are given in [11]:

1. A production rule must be *applicable* to a given graph in order for transformation to be possible. A rule is applicable if there exists a *matching* of the rule in the graph. In fact, there may be multiple different matchings of the same rule in the same graph.

2. Given a matching of a rule in a graph, the rule prescribes that certain nodes and edges are *deleted* from the graph and some nodes an edges are *created*, i.e., added to the graph.

3. Which nodes and edges are deleted and where new ones are added in the graph is determined relative to the matching; thus, in general, each matching gives rise to a different target graph.

4. The process of deletion and creation results in a new graph which is the target graph of the transformation.

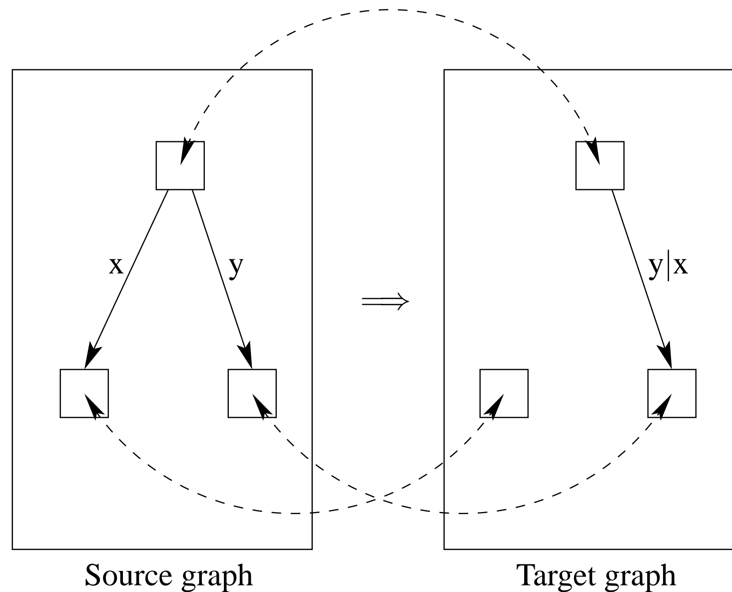

Figure 2.3: Example of a production rule

A set of production rules and an initial graph together is called a *graph grammar*. From this graph grammar it is possible to generate a whole system of reachable states. This is done by applying the production rules in all possible ways in which they are applicable. This system we call a Graph Transition System. In Section 2.3 we will see that the tool (GROOVE) can be used to generate this Graph Transition System.

In the remainder of this document we will follow the *single-pushout approach* for graph transformation (see [8, 3]) where the rules are enhanced with (certain kinds of) *negative application conditions* (see [5]).

2.3 GROOVE

GROOVE stands for GRaphs for Object-Oriented VErification. The GROOVE project aims to develop a toolset for analyzing and verifying object oriented programs. Currently GROOVE consists of an editor for creating graph production rules and a simulator for computing the graph transformations induced by a set of graph production rules. Also it is a complete framework for loading, saving, creating, and editing of graphs and graph production rules which can be used by other programs.

The graphs in GROOVE are very simple: the graphs consist of nodes, without labels, and directed edges that do have a (single) label. In previous figures you have seen nodes that have labels, but this is only a representation trick. The labels of nodes are really labels of self-edges, i.e. edges from the node to itself. One reason why proper node labels have been excluded is that this makes it more straightforward to encode patterns in which the label is irrelevant.

In GROOVE the source and target graph of a production rule are combined into one graph. To be able to do this special semantics is added to the graph:

- *Reader* nodes and edges. These nodes and edges need to exist in a source graph in order for the rule to be applicable, but will not be affected by applying the rule, so the reader nodes and edges will still exist in the target graph. Graphically these nodes and edges are depicted by solid black arrows and solid black nodes.

- *Eraser* nodes and edges. These nodes and edges must exist in the source graph for the rule to be applicable. After the rule is applied these element will be removed, so they do not exist in the target graph. The *eraser* nodes and edges are depicted by dashed blue arrows and blue double-bordered nodes.

- *Embargo* nodes and edges. These nodes and edges are forbidden to exist in the source graph for the rule to be applicable. The *embargo* elements are depicted by closely dashed fat red arrows and red double-bordered nodes.

- *Creator* nodes and edges. These nodes and edges will be created in the target graph when the rule is applicable: all other conditions are met. The *creator* elements are depicted by solid fat green arrows and nodes.

All the different elements are depicted in Figure 2.4. This rule is an example of how a production rule can look like in GROOVE. Almost all different elements are shown, except eraser and creator nodes.

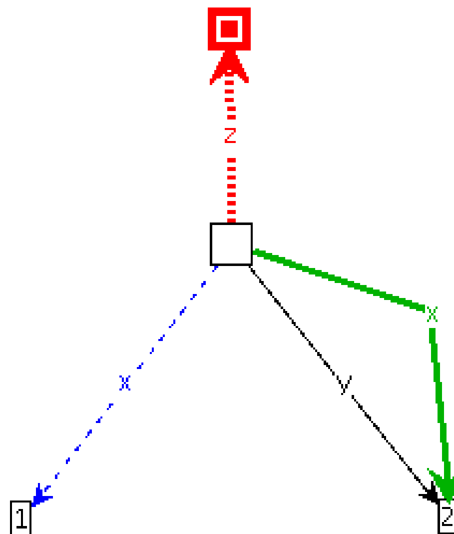

Figure 2.4: Example of a production rule in GROOVE representation

Groove input format

The visual representation of production rules in the previous section is the output format of GROOVE. To be able to generate and store production rules with the editor of GROOVE, we need a textual representation of this format. To be able to do this special *role prefixes* are introduced, listed in Table 2.1. For nodes, the role is indicated by a special self-edge labeled exclusively by the role prefix; for edges, the prefix is inserted in front of the edge label. By

using this special format also production rules can be seen as graphs with special labels. In Figure 2.5 the production rule of Figure 2.4 is given in the input format.

Prefix	**Meaning**
(nothing)	Reader node or edge
use:	Reader node or edge
del:	Eraser node or edge
not:	Embargo node or edge
new:	Creator node or edge

Table 2.1: Role prefixes in the input format for production rules

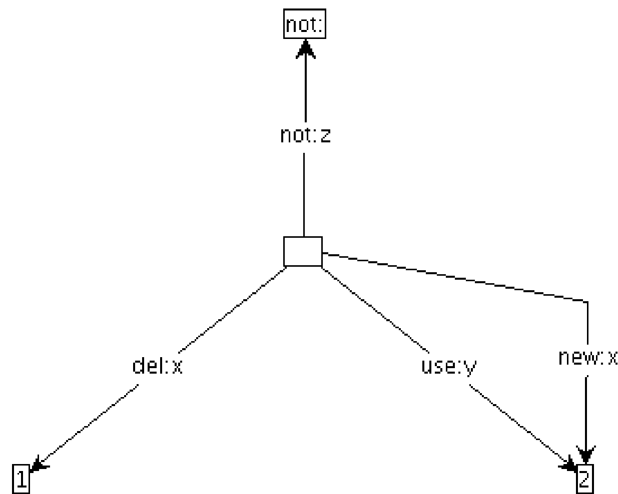

Figure 2.5: Production rule of Figure 2.4 in GROOVE input format

Graph Transition System

The simulator of GROOVE is able to generate a Graph Transition System. This Graph Transition System is generated by generating all possible transitions from a initial graph. An example of a initial graph is given in Figure 2.6; this is the starting graph of the example introduced in Chapter 3. By using the production rules generated by the translator, the simulator is able to calculate all reachable states of the system. These states are used to simulate all the states the Java program can reach.

In Figure 2.7 a part of a Graph Transition System generated by the groove simulator is depicted. The nodes labeled by *S0*, *S1* and so on are the states, the edges are the transitions between the states, labeled by the name of the production rule which produce the transition. The starting state is labeled by *S0*. Also can been seen the execution of a Java program is not always deterministic, different orderings of execution are possible. The total Graph Transition System of this example contains 111 states.

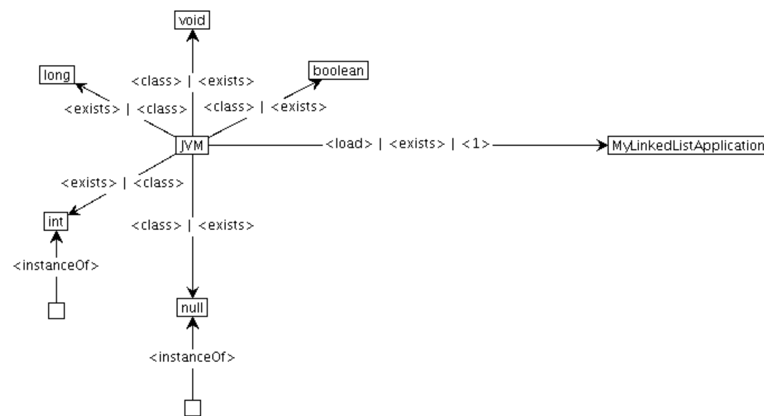

Figure 2.6: Example of a initial graph

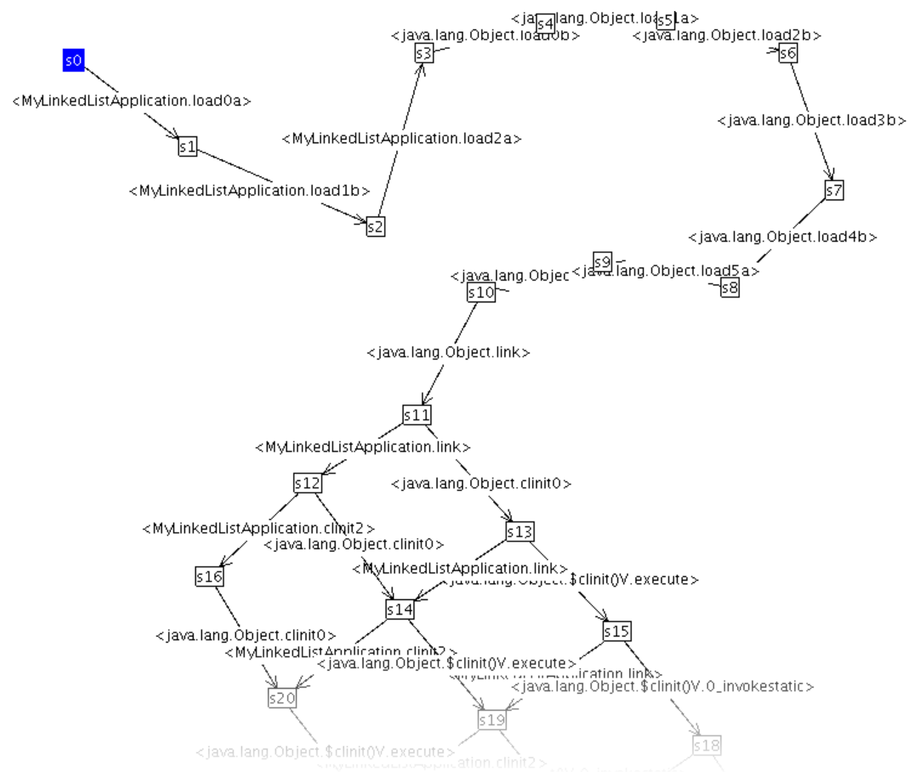

Figure 2.7: Part of a Graph Transition System

Chapter 3

Translating Java

Programs written in the programming language Java are compiled into a portable binary language called *byte code*. The Java byte code is the intermediate language of Java which can be executed by using a virtual machine on many different platforms. All different classes of a program are in separate files, each of these *.java* files will be compiled into separate Java class files. Such a Java class file contains all information about a class and all the byte code instructions.

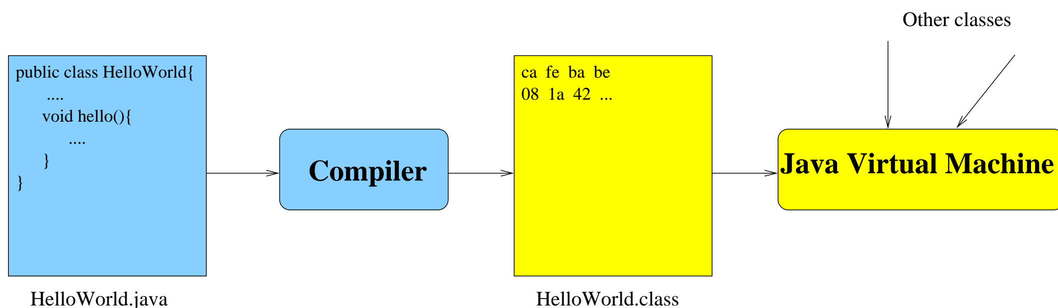

Figure 3.1: Compilation and execution of Java classes [2]

In Figure 3.1 the whole process of compiling source code and executing a Java class file is depicted. First the *HelloWorld.java* must be compiled into Java byte code. This Java byte code can be loaded, interpreted and executed by the Java Virtual Machine (JVM).

To translate a Java program into graph grammars there are two possible ways:

- Translate the source code, the *.java* files, into graph grammars
- Translate the byte code, the *.class* files, into graph grammars

In this paper we will focus on translating the byte code into graph grammars. The translation of source code into graph grammars is researched by A. Lozano Rodríquez, see [12].

The reason we want to be able to translate a java program by using both methods is because both methods have advantages. Translating the source code into graph grammars is not always

possible because the source files are not always available, for example if you are using other libraries. By using the Java compiler it is possible to transform every Java program into byte code. So if the translation from byte code to graph grammars is possible, it would be possible to translate every Java program to byte code and then to graph grammars. But this way it is harder to trace a bug back to the source code, the transitions between the run-time states correspond to JVM instructions, not Java statements. In the translation from Java byte code the names of local variables are lost. Also some structures of a program are lost, for example a *while loop* is implemented in byte code by a conditional jump and a jump back at the end of the *while loop*.

3.1 The Java Linked List example

Throughout this document we will use the same example to illustrate different aspects of the translation into graph grammars. The example used is a simple linked list, which can be extended to a double linked list. In a linked list, values are stored in nodes and these nodes are connected by pointers pointing to the next node. In a double linked list also a pointer is pointing to the previous node, this way you can traverse the linked list forward and also backwards. A picture of a single linked list is given Figure 3.2 and a double linked list is depicted in Figure 3.3.

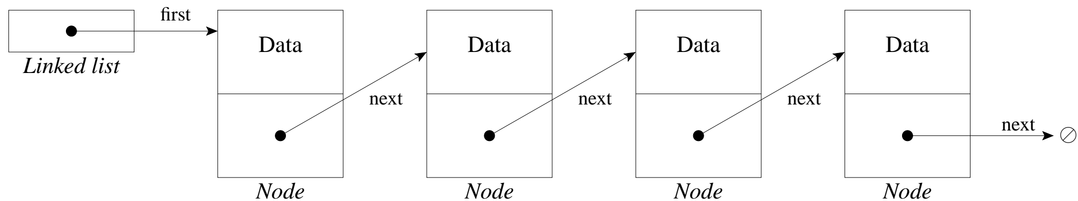

Figure 3.2: Example of a single linked list

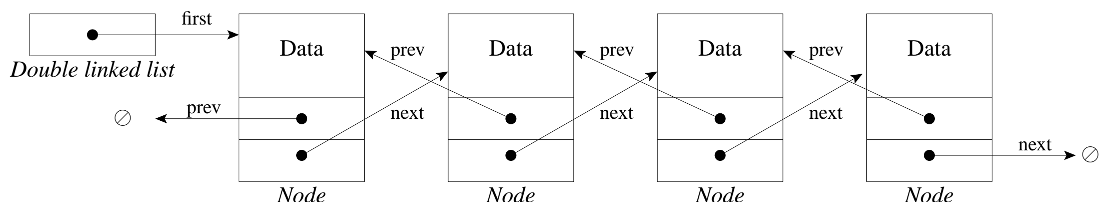

Figure 3.3: Example of a double linked list

There are three classes and two classes that extends two others:

- MyLinkedListApplication: Simple application that uses the linked list
- MyLinkedList: The list itself, with the add method to modify the list
- MyLinkedListNode: A node in the linked list, this contains the data of the list
- MyDoubleLinkedList: An extension to MyLinkedList to implement a double linked list

- MyDoubleLinkedListNode: An extension to MyLinkedListNode to implement a double linked list

In Figure 3.4 the UML diagram of the double linked list example is given. As can been seen, the single linked list is extended to a double linked list. In this example many important aspects of Java are used, for example: object creation, fields, field assignments, and inheritance. Using this example we can show how class loading, linking and initialising must be done, but also how methods and fields can be resolved in graphs.

The source code of this example is included in appendix A.

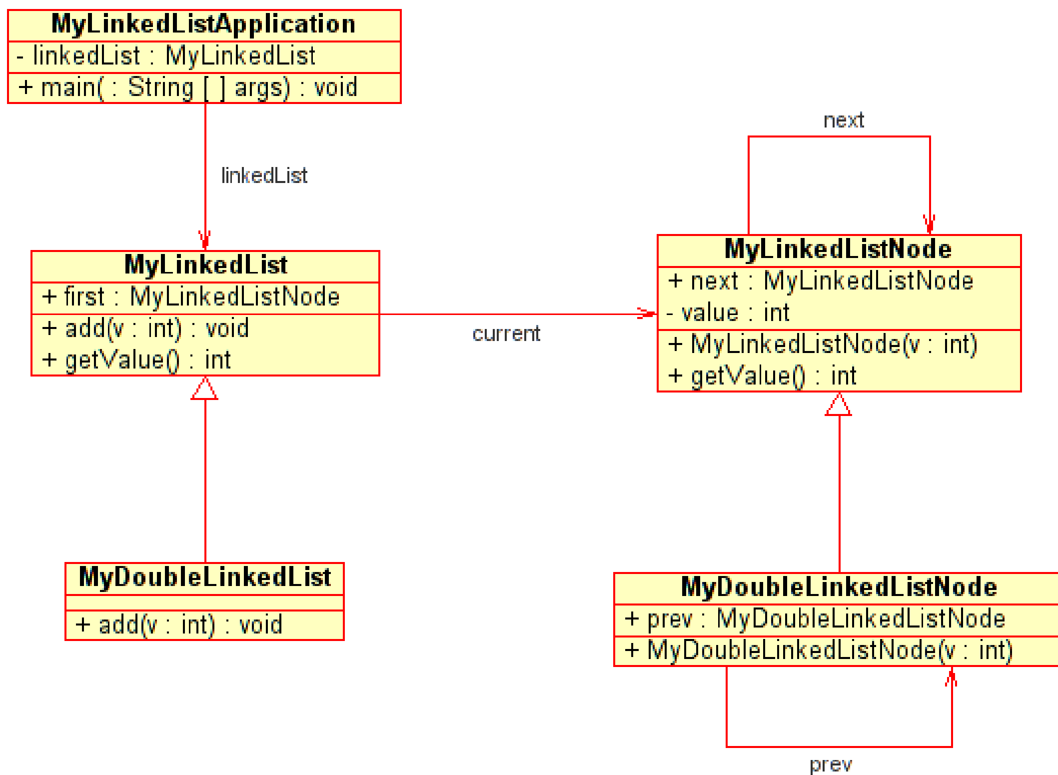

Figure 3.4: UML diagram of the Double Linked List example

3.2 The Java Virtual Machine

The JVM is an abstract computing machine. Like a real computer it has an instruction set and manipulates various memory areas. The JVM is implemented as a simple stack machine. Every method has its own memory space called a method frame. This method frame consists of a stack where values can be pushed on and popped from. Also every method has a number of local variables it can use; the local variables are numbered from 0 to 65535.

3.2.1 Java class file format

A Java class file contains all information needed for executing the class. This is not only the Java byte code instructions themselves but also information about the implemented interfaces, containing methods, fields etc. In Figure 3.5 a simplified overview of the Java class file is given. A complete overview of the java class file format is out of the scope of this document; see [7].

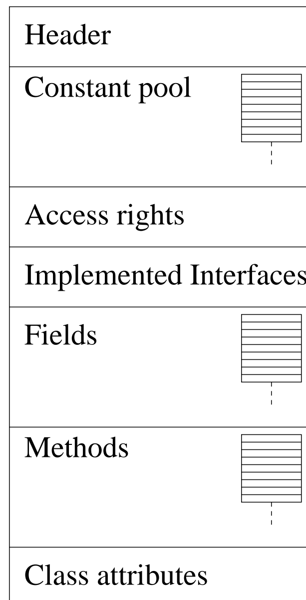

Figure 3.5: Java class file format [2]

The *header* of a class file contains a magic number identifying the class file format and the version number of the class file. The *constant pool* is a table of structures representing various sting constants, class and interface names, field names, and other constants that are referred to within the class file. Most of the time the constant pool is the largest portion of a class file, on average 60% of the class file is taken by the constant pool. The information in the constant pool is used to dynamically resolve the symbolic references to classes, fields and methods at run-time. The *access rights* are the access rights to this class encoded as a bit mask. *Implemented interfaces* is a list of the interfaces this class implements, actually it is an array with indexes to the *constant pool* and in the constant pool the names of the interfaces are given. *Fields* contains a list of fields declared in this class. *Methods* is a list of methods implemented by this class and also the method body, the JVM instructions, is located here. The last item of the class file format is *Class attributes*; this contains attributes of the class. The attributes defined in this part are ignored by the JVM; they only provide additional descriptive information. An example of an attribute is the SourceFile attribute which describes the source file of the class file.

To translate a class file into graph grammars only the *constant pool*, *access rights*, *implemented interfaces*, *fields*, and *methods* are used. The *header* information and the *class attributes* are ignored. This is not a restriction because the *header* is only used to verify it is an class file and the *class attributes* contains only extra information that is not needed for executing the class.

3.2.2 Byte code instruction set

At the moment the byte code instruction set consists of 212 instructions, of which 44 are reserved and may be used in the future. The byte code instruction set can be grouped, here we take the grouping used in [2]:

- *Stack operations*: Operations that control the stack, like *iconst_0* and *bipush*. These operations can push and pull values on and of the stack.
- *Arithmetic operations*: Operations that compute a result of two values, like adding two integers and subtracting two values. There are different instructions for different types. For example operations starting with *i* denote integer operations. Examples of operations in this class are *iadd* and *fmul*.
- *Control flow*: Operations that control the flow through the program. This are the branch instructions like *goto* and *if_icmpeq*. The change of the control flow can be conditional, *if_icmpeq* compares two integers, and if they are not equal the control flow is changed.
- *Load and store operation*: Operations that control the local variables. For example *iload* and *istore*.
- *Field access*: Operations that take care of the access to instance fields and static fields. For example *getfield* and *putstatic*.
- *Method invocation*: Operations that invocate methods. The instructions for invoking different kinds of methods are: *invokestatic*, *invokevirtual*, *invokeinterface*, and *invoke-special*.
- *Object allocation*: Operations that create new instances of objects. For example *new* and *newarray*.
- *Conversion and type checking*: Operations that convert types to other types, or check validity of a type cast. Examples are: *f2i*, *checkcast*, and *instanceOf*.

All the operations consist of a one-byte opcode specifying the operation to be performed, followed by one or more operands. The operands supply the arguments of data that are used by the operation. Most instructions have a fixed number of operands, a few have a variable number of operands, for example the *lookupswitch* and *tableswitch* which are used to implement the switch() statement, since the number of case statements is variable.

In Listing 3.1 the add method of the class MyLinkedList.java is given. On line 2 a new node is created, the *next* pointer of the new new node is set to the current first node and on line 4 the new node is made the first node.

```
1 public void add(int v){
2     MyLinkedListNode newNode = new MyLinkedListNode(v);
3     newNode.next = first;
4     first = newNode;
5 }
```

Listing 3.1: Add method in MyLinkedList.java

```
public void add(int arg1)
Code(max_stack = 3, max_locals = 3, code_length = 23)
0:    new         <MyLinkedListNode> (2)
3:    dup
4:    iload_1
5:    invokespecial MyLinkedListNode.<init> (I)V (3)
8:    astore_2
9:    aload_2
10:   aload_0
11:   getfield      MyLinkedList.first LMyLinkedListNode; (4)
14:   putfield      MyLinkedListNode.next LMyLinkedListNode; (5)
17:   aload_0
18:   aload_2
19:   putfield      MyLinkedList.first LMyLinkedListNode; (4)
22:   return
```

Listing 3.2: Add method in MyLinkedList.java

When the above piece of code is translated by the compiler, it is translated into JVM instructions. In Listing 3.2 the corresponding JVM instructions are given. This listing is generated by using the Byte Code Engineering Library (BCEL), see section 5.1.1 for more information.

Most of the Java statements are translated into one or more JVM instructions. Java statements that declare variables are not translated into JVM instructions; this information is only used by the compiler.

In front of each JVM instruction the program counter is given (PC). Each instruction has the length of the instruction it self plus its operands. The instruction *new* begins when the program counter is zero, and the instruction *dup* begins when the program counter is 3, this is because the length of the new instruction is 3. This program counter defines in what order the JVM instructions are executed. By changing the program counter the flow of the program is changed; this is used for conditional statements and jumps.

At first the translator will not be able to translate all JVM instructions, only a subset. When a JVM instruction is being translated which is not supported by the translator, the translator will give a warning. For an overview of the implemented instructions see Appendix D. For more information about all instructions see the Java Virtual Machine Specification [7].

Chapter 4

Representation

Before a program can be translated into a graph grammar, we must know how to represent all different aspects of the program in a graph and graph production rules. This is also very important because the translation of Java source files into graph grammars (done in a different project, see [12]) and the translation of the Java byte code into graph grammars needs to be compatible, so you can use the two results together. Therefore a Meta Model is developed in which all different aspects are represented. See section 4.2 for details about the Meta Model.

4.1 Representation of different Java aspects

The following paragraphs will give an overview of how different aspects of the Java programming language are represented in graphs and graph production rules. The names of nodes used in these paragraphs is summerised in Table 4.1, also a short description is given. For more information see the following paragraphs and Appendix C

Node name	Short description
JVM Node	Represents the Java Virtual Machine
Interface Node	Represents an interface
Class Node	Represents a class
Object Node	Represents an object
Method Signature Node	Represents the declaration of a method, its signature
Method Frame Node	Represent an instance of a method
Instruction Order Node	Represents a node for keeping track of the instruction order
Stack Node	Represents a part of a stack

Table 4.1: Nodes overview

4.1.1 Java Virtual Machine

Because we want to simulate a program that is running inside the Java Virtual Machine also the JVM itself needs to be represented in a graph. The label of the JVM node is ”JVM”. This JVM node simulates the behaviour of the JVM. For example it is the JVM that invokes the first

main method of a program, so in the starting graph this invoke is done. This starting graph, the first state of the system, is called the initial graph of the graph grammar.

This JVM node also keeps track of the loading state of each class; this is represented by different edges from the JVM node to the classes. For more information about loading, linking and initialisation of classes see Section 4.3.

The current active method can also be found from the JVM node. If a method is active there will exist an $\langle active \rangle$ edge to this method from the JVM node.

4.1.2 Types and values

Classes

A class definition specifies a new reference type and gives its implementation. This reference type has to be represented in a graph. A class is represented by a node with its fully qualified name. The fully qualified name contains the name of the package the class is in and the name of the class. Each class is only loaded once in the whole system, so it is unique. Figure 4.1 depicts an example of a class. More on loading, linking, and initialising of classes can be found in Section 4.3.

java.lang.Object

Figure 4.1: java.lang.Object as a Class node

Classes can derive implementation from another class, when this is the case it is called the direct superclass of the current class. In fact every class has a superclass, if it is not explicitly specified then it is Object. Only Object itself does not have a superclass. A superclass is represented in a graph by an edge $\langle super \rangle$ from the subclass to the superclass, depicted in Figure 4.2. When a class is a subclass of another class it can inherit the methods, see Section 4.1.3, and the fields, see Section 4.1.6.

Interfaces

Interfaces are also represented by a node with as label its fully qualified name. An interface must be implemented by a class because it has no implementation. If a class implements a certain interface it will have an edge $\langle implements \rangle$ from the class node to the interface node.

Types

Each type is represented by a class node, so also primitive types are represented by a class node. Also *null* and *void* are represented by class nodes. The difference between primitive types and other types is that the primitive types are already in the initial graph.

In Figure 4.3 the null value is depicted and the variable *first* with a reference to this null value.

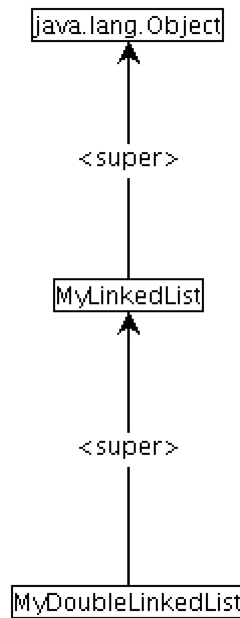

Figure 4.2: Chain of superclasses

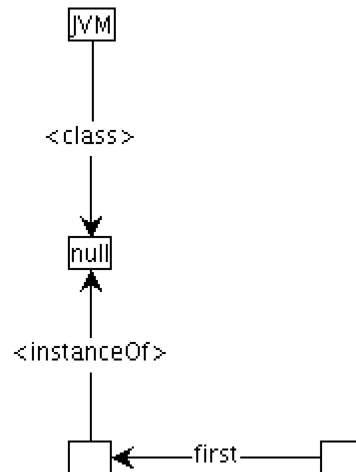

Figure 4.3: A reference to the null value

Instances

Instances of types are represented by object nodes. For example if a new object of class MyLinkedList is created, an object node is created with an edge $\langle instanceOf \rangle$ to a corresponding class node. This is depicted in Figure 4.4.

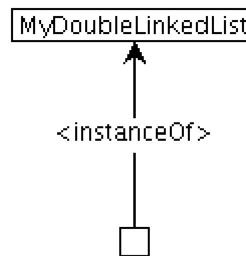

Figure 4.4: instance of MyDoubleLinkedList

Primitive values

The values of primitive types are not represented. For example the value of an integer is not used, we abstract from the exact values. Each primitive type (except boolean) will only have one value, and so only one instance in the graph. This is done to reduce the possible states of a program and to reduce the size of a graph representing a state. Also the computation of a arithmetic operations becomes very simple, for example if two integers must be summed, these two integers are replaced by a new integer; the exact value does not matter. The reachable states of a program are reduced because if a value of a primitive type is changed it is not a new state; for example a *for* loop that changes the value of an integer thirty times, it would normally result in thirty different states. Without the exact values this can be reduced to one state. The size of the graphs are reduced because each value have only one value.

This however can also introduce incorrect states, states that cannot be reached by the program. For example an if statement will always result in two different outcomes, one for when the statement is true and one for when the statement is false. So if you analyse a program, and you have found a state that is wrong, for example a deadlock situation, you will have to check if this state can really be reached by the program.

This choice make it impossible to analyse if a loop in a program terminates. This is a great disadvantage of this choice, so maybe in the future an option to use the exact values of primitive types must be investigated.

A special case is the *boolean* type, we have not abstracted from its value because it can only have two values: true and false. Figure 4.5 depicts the two possible instances of boolean.

4.1.3 Methods and constructors

Methods

A method contains the executable code that can be invoked. In a graph a method is represented by a node with its signature as its label, called a Method signature node. This signature consists of the name of the method and the parameters of the method. For the signatures the Java Native Interface (JNI) specification [6] is used.

The signature for a method has the following form:

method-name (argument-types) returned-type

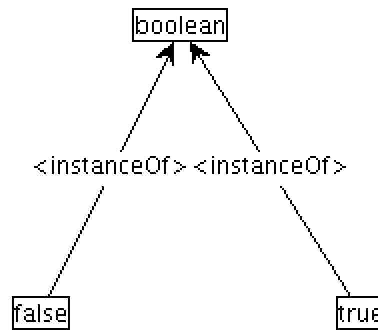

Figure 4.5: instances of boolean

These types are the Java types of the arguments, encoded by using Table 4.2. The null type is not in this list because it cannot be in a signature of a method.

Java Type	**Signature**
boolean	Z
byte	B
char	C
short	S
int	I
long	J
float	F
double	D
void	V
type []	[type
reference	Lclass-name

Table 4.2: Java types as signatures

For example the signature for a main method is: *main(Ljava/lang/String;)V*. When looking at the signature you can see the method expects an argument from the class *java.lang.String* and has a type *void* as return value.

In Figure 4.6 the methods of class *MyLinkedListApplication* are given. As you can see the class has two methods: $\langle init \rangle$ and *main*, these are the two methods that are declared in the class itself.

Constructors

The method $\langle init \rangle$ is the constructor of the class. All classes have an $\langle init \rangle$ method even if this is not defined in the source code. The $\langle init \rangle$ method is invoked right after a new instance of the class is created. Classes themselves can also have a (static) initializer or an initialising method, called a $\langle clinit \rangle$ method. In this method all static's are initialised and all static initializers are located.

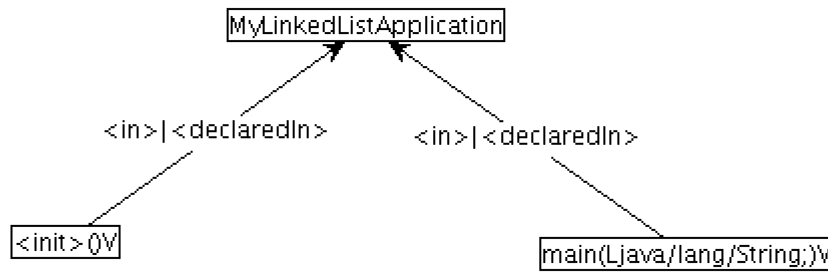

Figure 4.6: Methods of MyLinkedListApplication

Inheritance

Classes also have methods that are derived from the superclasses. Methods have (at least) two edges, an $\langle in \rangle$ and a $\langle declaredIn \rangle$ edge, to be able to find the class where the real implementation of the method is, and to find the class where the method can be used. The $\langle declaredIn \rangle$ is pointing to the class where the real implementation can be found, the $\langle in \rangle$ edge is pointing to the class where the method can be invoked. In fact there can be multiple $\langle in \rangle$ edges from one method to multiple classes, to the class where the method is declared and to all the inheriting classes if the method can be inherited and the subclass is not overriding the method.

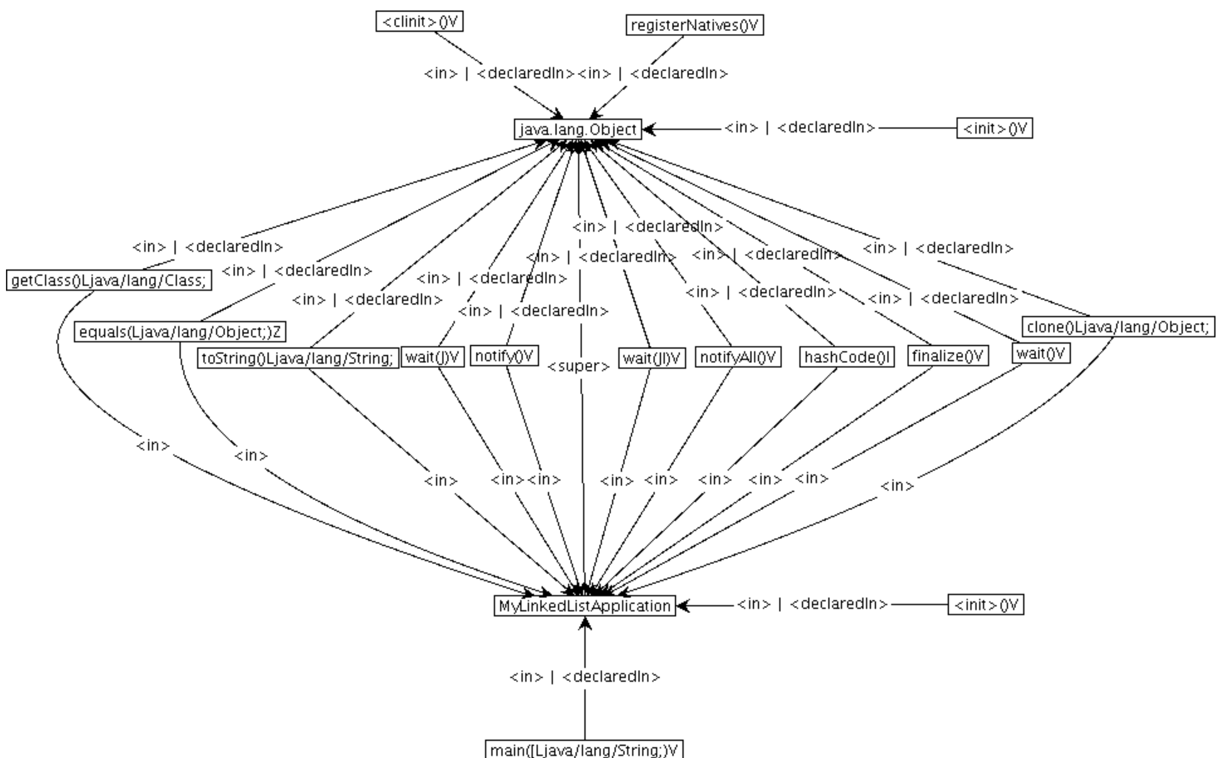

Figure 4.7: Methods of MyLinkedListApplication after linking

```
1 0:    new       <MyDoubleLinkedList> (2)
2 3:    dup
```

Listing 4.1: The *dup* instruction that modifies the stack

In Figure 4.7 also the inherited methods are depicted. This state depicts the state after *MyLinkedListApplication* has been linked. All inherited methods have an $\langle in \rangle$ edge to *MyLinkedListApplication*. The method $\langle init \rangle$ is overridden in the subclass, therefore it is twice in the figure and it has also two different $\langle declaredIn \rangle$ edges. Also the methods $\langle clinit \rangle$ and *registerNatives* are not inherited, this is because these methods are not accessible methods of the superclass.

The reason for having two edges, the $\langle declaredIn \rangle$ edge and $\langle in \rangle$ edge, for the method lookup in a graph is described in Section 4.4.

Instances

When a method is invocated a new instance of the method will be created, called a Method Frame Node. Every Method Frame Node has his own stack (see Section 4.1.4) and program counter (see Section 4.1.5). From the new Method Frame Node there will be an $\langle instanceOf \rangle$ edge to the corresponding Method Signature Node. The parameters of the invocation of the method will be passed as local variables to the new Method Frame Node. The local variables are depicted as edges from the Method Frame Node to the value of the parameter and as label the number of the localvariable: $\langle number \rangle$. If the method returns, a $\langle return \rangle$ edge will be created from the return value to the calling method.

For more information about the invocation and execution of methods see Section 4.4.

4.1.4 Stack

To be able to simulate all the Java byte code instructions, also the stack of a method has to be simulated. Each Method Frame Node has a stack. The stack is simulated as a simple linked list: nodes connected to each other by $\langle previous \rangle$ edges and the value of a stack location is connected by a $\langle value \rangle$ edge. The current, and last, value can be found by the $\langle current \rangle$ edge. An example of the stack and an instructions that modifies the stack is depicted in Listing 4.1 and Figure 4.8. The *dup* instruction duplicates the last value on the stack.

4.1.5 Control Flow

To define the order of the different JVM instructions in a method the JVM uses a program counter register. This program counter (PC) is also used for branch instructions so loops and jumps can be implemented; the branch instructions change the value of the PC so the JVM continues at the right address (line of code). This way of ordering is also used for the translation to graph grammars. Each Method Node has an edge called $\langle PC \rangle$ to a node with as label the current value of the program counter. The value of the program counter is a number; this value is the same value as the program counter of the JVM instructions.

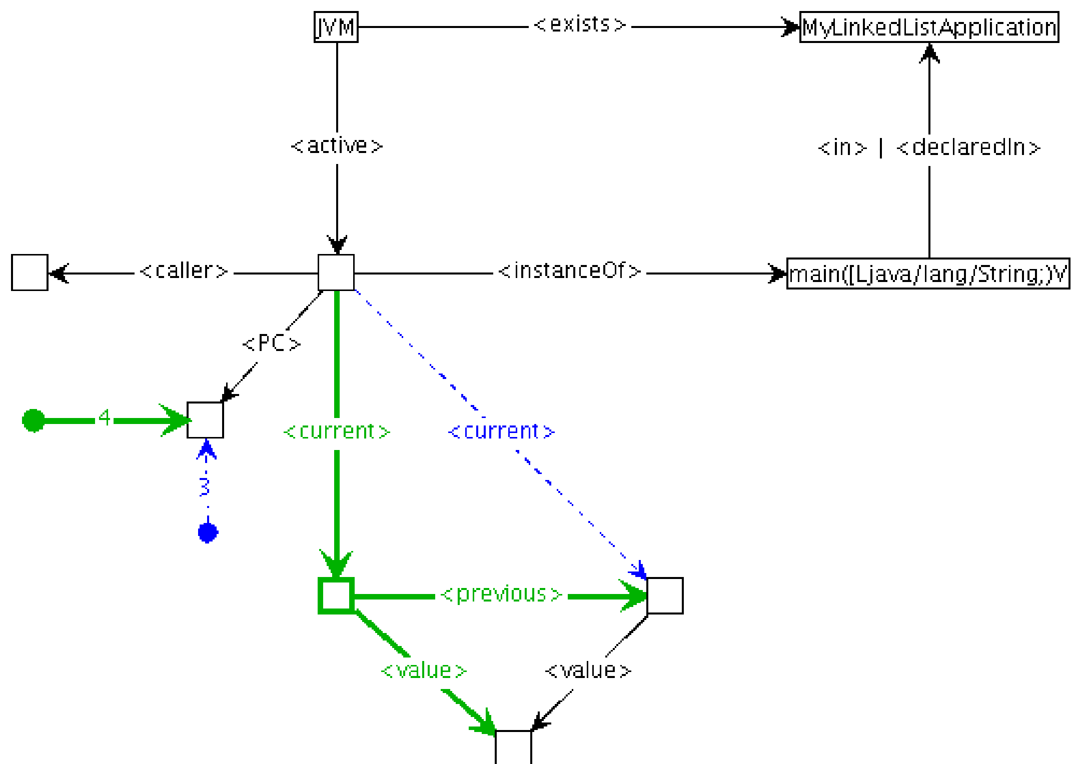

Figure 4.8: The *dup* instruction that modifies the stack

The $\langle active \rangle$ edge defines which method is active at the moment. When a other method is invocated or the method returns, the $\langle active \rangle$ edge is deleted and a new one to the right method is created. This way only one method can be active at the same moment. At the moment threads are not supported; there must be multiple $\langle active \rangle$ edges when threads are supported, one for every thread.

In Listing 4.2 an example of a branch instruction in byte code is given. When the PC is 21, the instruction *ifnull* will be executed. This instruction will look if the value on the stack is *null*. If this is false the PC will be raised to 24, so extra code is executed. When it is true the PC will be raised to 35 and the extra code is not executed. These values of the PC are also used in the graph production rules, so each production rule is only applicable when the PC has the right value. The rule for the case the *ifnull* is true is depicted in Figure 4.9

4.1.6 Fields

There are two kinds of fields: *class variables* and *instance variables*. A *class variable* is a variable that is declared static in the source code. If a field is a static field, there exists exactly one at any time, even if there are many (or zero) instances of the class. This field starts to exists when the class is initialised, see chapter 4.3.3. An *instance variable* is created when a new instance of the class is created.

A field is represented by an edge to its value. If it is an *class variable* the source of the edge is a Method Signature Node and if it is an *instance variable* then the source is a Method Frame

```
1  18:   getfield      MyDoubleLinkedList.first LMyLinkedListNode; (6)
2  21:   ifnull        #35
3  24:   aload_0
4  25:   getfield      MyDoubleLinkedList.first LMyLinkedListNode; (7)
5  28:   checkcast     <MyDoubleLinkedListNode> (2)
6  31:   aload_2
7  32:   putfield      MyDoubleLinkedListNode.prev
       LMyDoubleLinkedListNode; (8)
8  35:   aload_0
```

Listing 4.2: Example of branch instruction

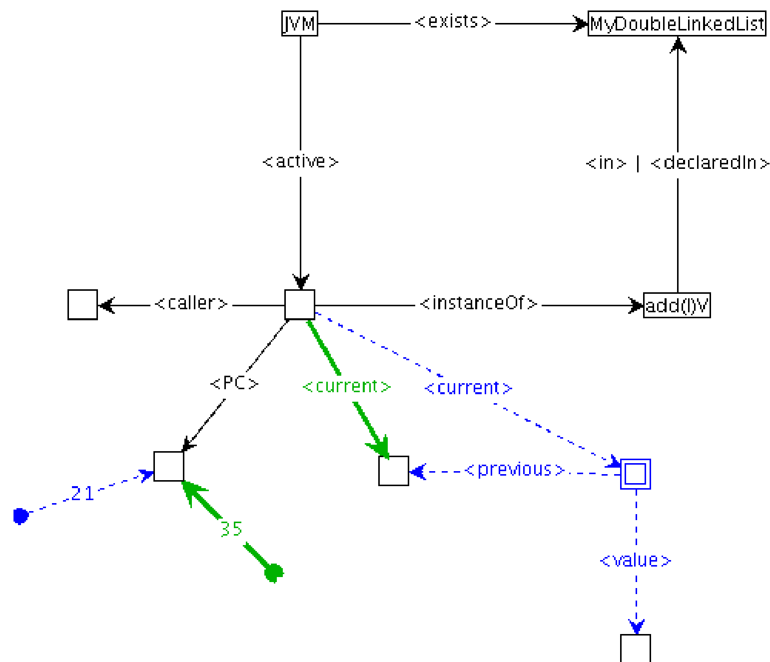

Figure 4.9: An example of the *ifnull* branchinstruction for the case it is true

Node. The label of the edge is the name of the class the field is declared in combined with the name of the field. The class name is in the label because if a class is extended, the subclass also inherits all the accessible fields of the superclass, and a variable in the subclass can have the same name as in the superclass. Both variables can be accessed; the variable declared in the subclass can be accessed by just the name of the variable, the variable declared in the superclass can be accessed by the word super before the variable, for example *super.testvar* or *((superclass) this).testvar*.

In Figure 4.10 an example is given of an *instance variable*; this is the variable *first* declared in the class *MyLinkedList*. The current value of *first* is a *null* value. In this example a lot other edges have been omitted.

Another way this could have been implemented is to have superobjects: if an instance of a class is created also instances of the superclasses of are created. This way the fields of the

superclasses could be connected to the superobjects. This solution is not chosen because this is not the way the JVM implements it, and it is harder to look up the right field.

Also another way fields could be implemented is to have fields as nodes, with an edge to the Methods Signature Node or Method Frame Node it belongs to and a $\langle value \rangle$ edge pointing to the value. The advantage of this method is that also with a $\langle type \rangle$ edge it is possible to keep track of the type of a field. But this method is not chosen because we don't want to do type checking with our model checker and the chosen solution reduces the amount of nodes and edges.

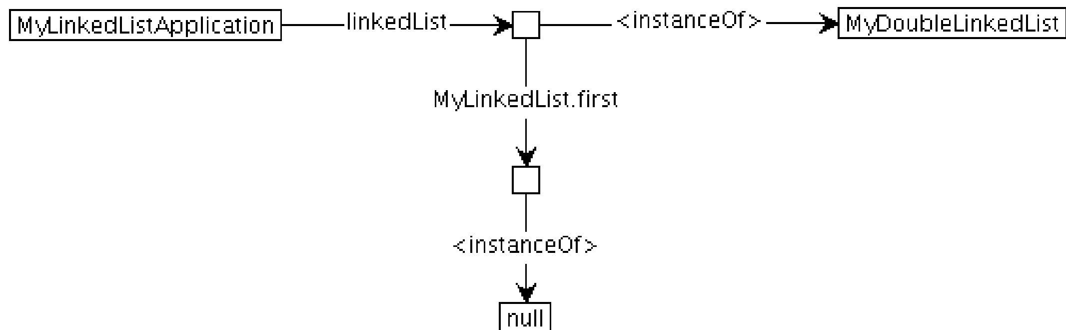

Figure 4.10: An example of a *instance variable*

4.2 Meta Model

Using the specifications of Section 4.1 we are able to make a model of the runtime state of every Java program. This model is also used for testing the graph grammar: every state of the simulation of a program must satisfy this model, if this is not the case there must be something wrong. This model is called a Meta Model because it is a model for all states of every Java program. In Figure 4.11 the model is given. In Table 4.3 a short description of the sets of edges is given and in Table 4.4 the names of the nodes are collected.

Name	**Description**
Status edges	Different edges which state the loading status of a class (see Section 4.3): $\langle exists \rangle$ indicates the class exists in the system $\langle load \rangle$ indicates the class must be loaded $\langle loading \rangle$ indicate the class is being loaded $\langle link \rangle$ indicates the class must be linked $\langle init \rangle$ indicates the class must be initialised $\langle initing \rangle$ indicates the class is being initialised $\langle class \rangle$ indicates the class is loaded, linked, and initialised
Identifiers	A set of names of the variables, the field names.

Table 4.3: Description of set of edges in the Meta Model

For an complete description of all nodes and edges see Appendix C.

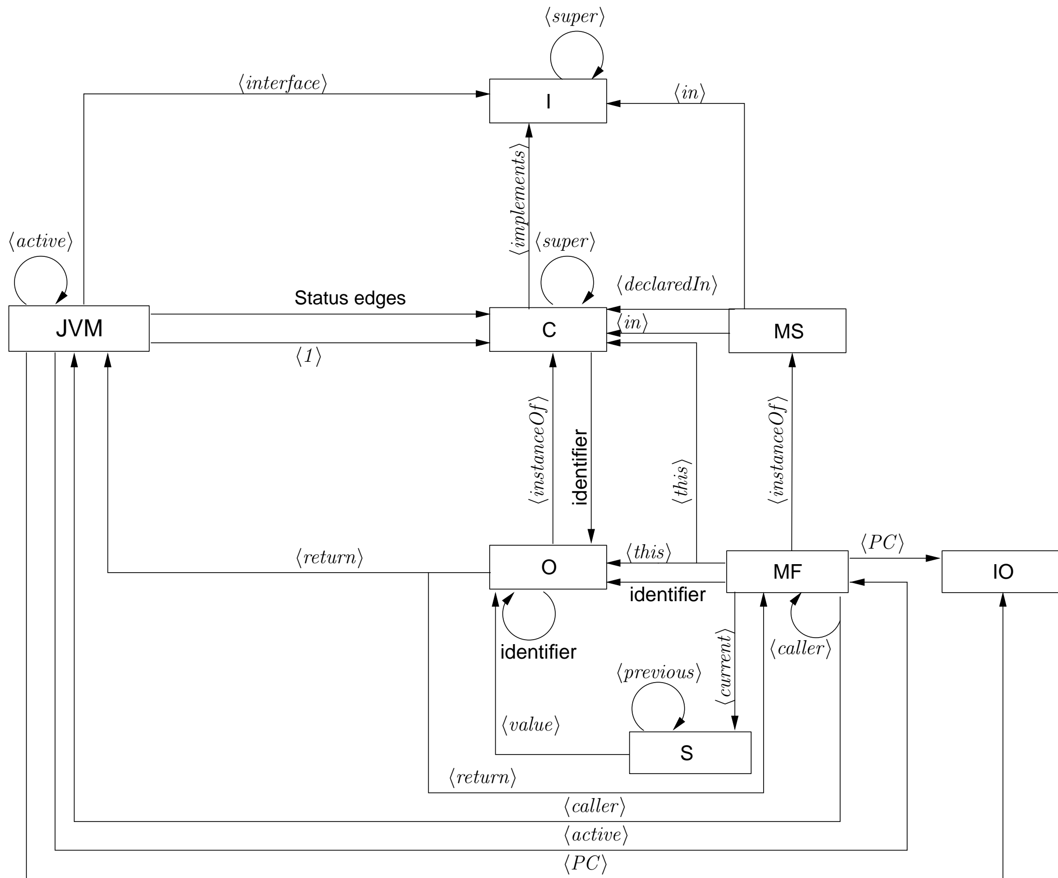

Figure 4.11: Meta Model of a Java program at runtime

Node	Label	Abbreviation
JVM	JVM	JVM
Interface	Name of the interface	I
Package	Name of the package	P
Class	Name of the class	C
Object	None	O
Method Signature	The method name followed by its signature	MS
Method Frame	None	MF
Instruction Order	A number	IO
Stack Node	None	S

Table 4.4: Nodes definition [12]

4.3 Loading, linking and initialisation

Before classes can be used, they must be loaded, linked and initialised. This process must always been done before a class can be used for the first time.

Loading This is the process of finding the binary form of a class or interface type with a particular name. In graph representation this will result in adding a Class Node to the graph.

Linking This is the process of preparing this binary format for execution. This involves creating the static fields for a class or interfaces and initialising such fields to the standard default values. Also references to other classes are resolved. In a graph representation this process will result in adding the right edges to other classes and methods.

Initialisation The *initialisation* process consit of executing the static initializers and the initializers for static fields.

The following paragraphs will describe what must be done in each step in more detail. In Table 4.3 an overview of the different status edges used in the following paragraphs is given.

4.3.1 Class loading

The loading of classes is done by the JVM when a class is used for the first time, so each class is loaded once. The simplest solution to simulate this process is to have separate load rules for each class. A load rule will create the Class Node when it is not there. The disadvantage of this is that there is no particular order in loading the classes and all possible orderings will be simulated, resulting in too many states to simulate this.

To be able to simulate the loading of the classes, the class loading must be done in a predefined order. When a class is being loaded, also the classes the class depends on are loaded; the loading of these other classes is done in a predefined order. For this reason the JVM will have a program counter. To indicate a class is being loaded, a $\langle loading \rangle$ edge is created from the JVM Node to the Class Node. This edge also guarantees only one class is loaded at the same time.

Before a class is loaded a rule will check if the class does not already exist. This is done by checking there is an $\langle exists \rangle$ edge from the JVM to a class node with the right name. If it does not exist already this means the class must be loaded: a new Class Node and an edge called $\langle load \rangle$ from the JVM to the new Class Node will be created. When all classes a class depends on are loaded the $\langle load \rangle$ edge will be replaced by a $\langle link \rangle$ edge to indicate that the linking phase of this class. Also the program counter of the JVM, and the $\langle loading \rangle$ JVM edges are removed.

In Figure 4.12 an example is given of the first rule of a loading sequence. If the needed class is not loaded already, a new node is created with an edge to indicate the class needs to be loaded. Also the program counter is created. This rule can only be applied when no other class is loading, indicated by a $\langle loading \rangle$ edge. If the needed class already exists another rule can be applied: this rule will only create the program counter and the $\langle loading \rangle$ edge. This rule is

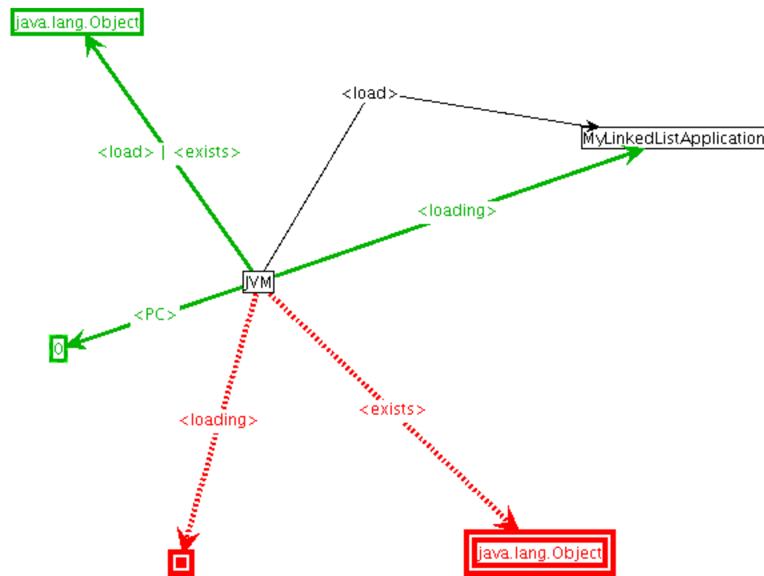

Figure 4.12: Example of first rule of a loading sequence

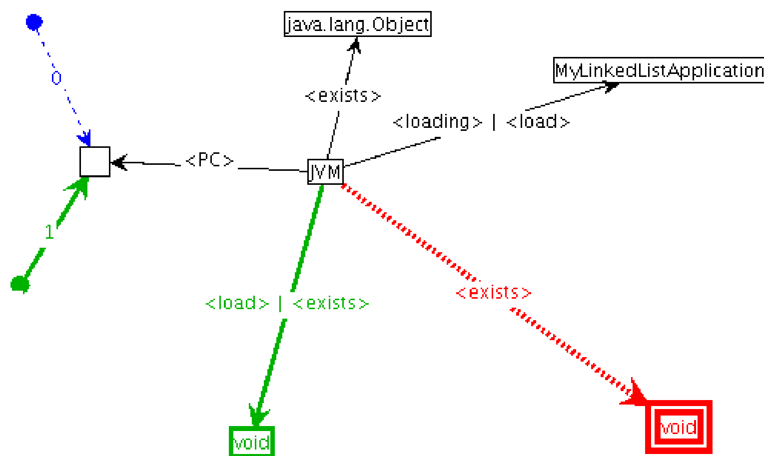

Figure 4.13: Example of next loading rule

necessary to continue the load process. There will always be two transformation rules, one for if the class does not already exists and one for if it already exist.

In Figure 4.13 the next rule is presented: if the needed class doesn't already exist it is loaded, also the program counter is increased.

In Figure 4.14 the last rule of the loading sequence of this class is presented. This rule can be applied if the last needed class is already loaded by the JVM. After this rule the linking of the class can begin, this is indicated by the new $\langle link \rangle$ edge.

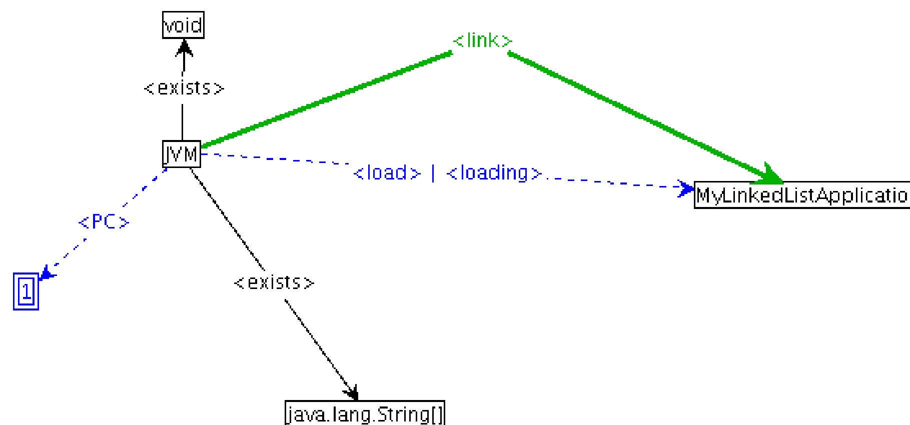

Figure 4.14: Example of last loading rule

4.3.2 Class linking

The linking phase of a class expect all classes it depends on are loaded already: super class, interfaces, types of method parameters, types of static fields, and return values. To ensure this, all classes it is depending on will be in this rule.

During this phase all needed edges to the other classes are created. Also the Method Signature Nodes of the class are created with $\langle in \rangle$ and $\langle declaredIn \rangle$ edges. When a method is inherited from a super class, only a new $\langle in \rangle$ edge is created from the already existing Method Signature Node.

The linking phase will also remove the $\langle link \rangle$ edge and replace it with an $\langle init \rangle$ edge to indicate the initialisation phase can start. An example of a link rule is depicted in Figure 4.15

4.3.3 Class initialisation

This is where the class itself will be initialised. Normally the JVM will initialize a class when it is used for the first time. Because this is not very easy to simulate we have chosen to initialize a class right after the linking phase; this doesn't change the behaviour of a program.

Initialisation of a class or interface consists of invoking its static initializers and initializers for the static fields declared in the class [7]. If a class has static initializers and initializers for the static fields the class has a method *clinit()*. If the class has a method *clinit()* this will be be invoked: an $\langle active \rangle$ edge from the JVM to it and a $\langle caller \rangle$ edge from it to the JVM will be created. Also an edge from the JVM to the Class Node is created to indicate the class is being initialised. This edge is also used to detect when the initialisation of a class is finished: when a *clinit()* method returns, it will create an $\langle active \rangle$ edge from the JVM to the caller Node, in this case also the JVM Node, so the rule in Figure 4.16 can only be applied when a certain *clinit()* is finished and the right class will get an $\langle class \rangle$ edge to represent it is loaded, linked, and initialised. An example is in Figure 4.17.

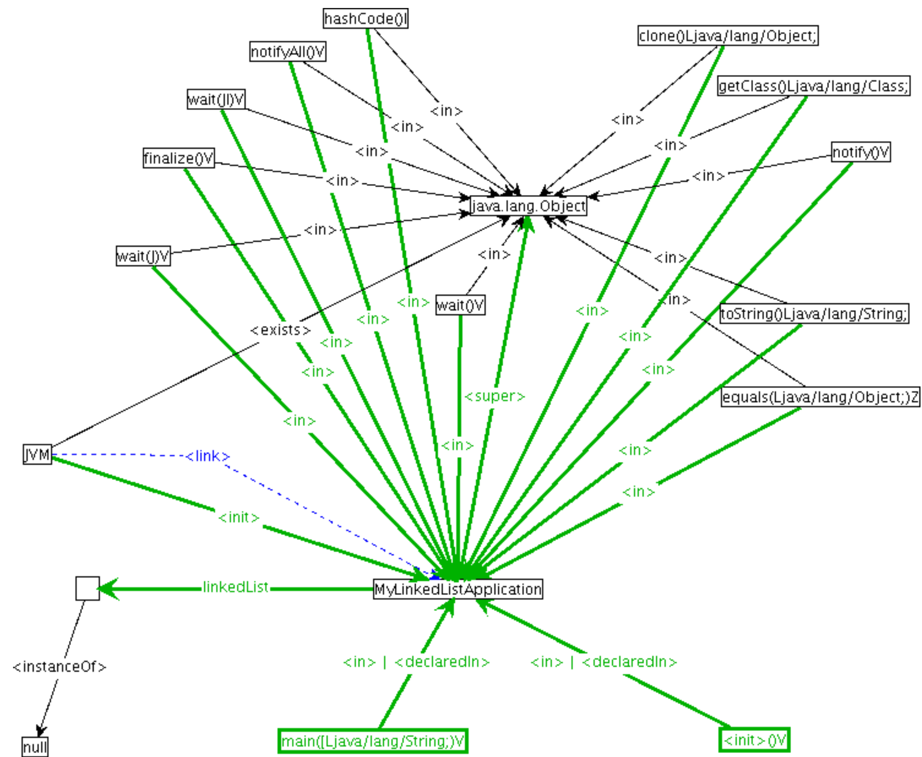

Figure 4.15: Example of a link rule

4.4 Method invocation and execution

4.4.1 Method invocation

The JVM has four different kinds of method invocation instructions:

- invokeinterface: Invoke interface method
- invokespecial: Invoke instance method; special handling for superclass, private, and instance initialisation method invocations
- invokestatic: Invoke a class (static) method
- invokevirtual: Invoke instance method; dispatch based on class

From these four different invoke instructions two will result in the same graph transformation rules: invokeinterface and invokevirtual.

invokeinterface and invokevirtual

The difference between invokeinterface and invokevirtual is that the method of invokeinterface is declared in an interface, but the implementation is in a class. The method of a invokevirtual is also implemented in a class, so the method to be invoked can be found in the same way.

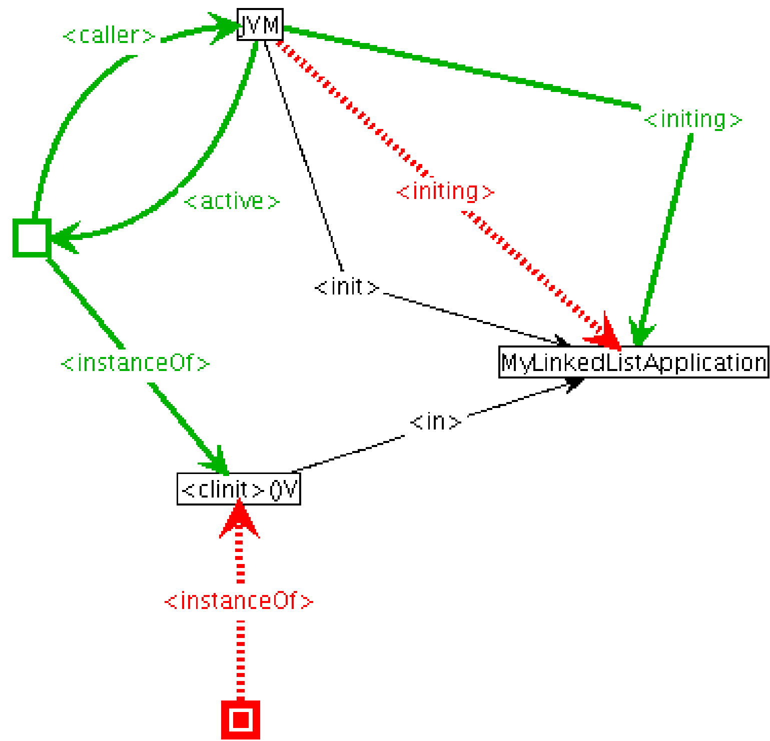

Figure 4.16: Example of a initialisation rule: invoke clinit()

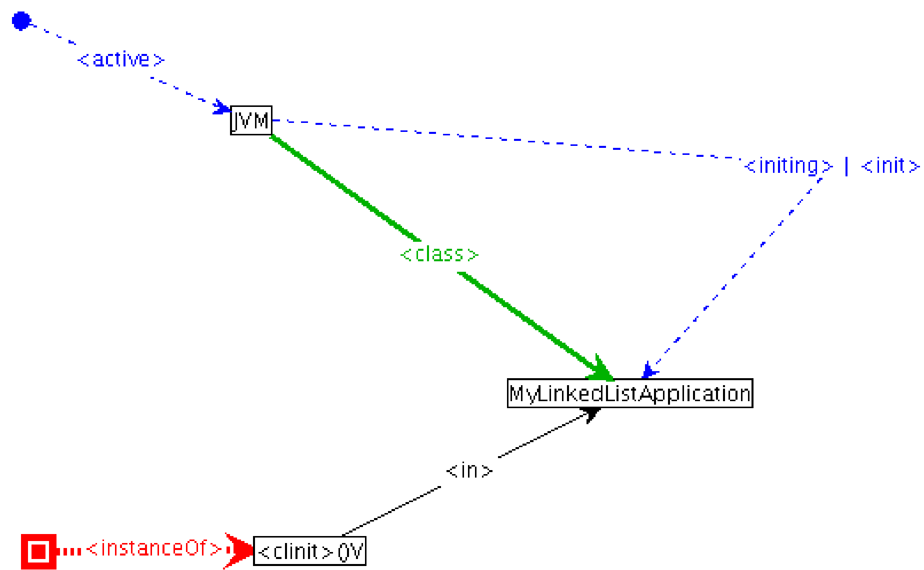

Figure 4.17: Example of a initialisation rule: after return of clinit()

If a method is invoked the following must be done:

1. Create a new Method Frame Node, this is the instance of the method that is invoked.
2. Create a $\langle instanceOf \rangle$ edge to the right Method Signature Node.

   When this instruction is executed, there will be a value on the stack, this reference value will point to an object. The $\langle instanceOf \rangle$ edge is pointing to the corresponding class. This class will have an incoming $\langle in \rangle$ edge from the method we want to invoke. Because the name, actually the whole signature, of the method is known, the right method of the class can be found.
3. Create a $\langle this \rangle$ edge to the right Object Node

   This is the Object Node that is the reference of the invoke instruction, so this is the value that was on the stack.
4. Create edges from the new Method Frame Node to its arguments.

   Arguments are passed to the invocated method as local variables, these local variables are numbered. The first local variable, $\langle 0 \rangle$, is always the reference to the object, the $\langle this \rangle$ edge. The local variables 1 to 65535 are the arguments. These arguments were on the stack of the method that invoked the other method, and are popped of from this stack.
5. Create a $\langle caller \rangle$ edge from the new Method Frame Node to the active method

   By creating this edge, it is known where to return, when the method is ready.
6. Delete the current $\langle active \rangle$ edge and create a new one to the newly created Method Frame Node

   This will make the invoked method the active method. The caller method will wait till the invoked method returns. When the invoked method returns, the caller will also get its $\langle active \rangle$ edge back, so it can continue execution.

In Figure 4.18 an example of the above steps is given. This is the invocation of the method *add* in the linked list example, which will add a new node to the linked list. As argument it expects an integer to be added to the list. For the transformation the types of the arguments are not important, only the number of arguments. In this case it are two arguments, the first argument is always the reference to the object itself. For this example it does not matter if it is a double linked list or not, the method to be invoked is dependent on the type of the object where the reference value is pointing to. If it is of type *MyLinkedList*, the *add* method of *MyLinkedList* is invoked, if it is of type *MyDoubleLinkedList*, the *add* method of *MyDoubleLinkedList* is invoked.

invokespecial

The difference between the invokespecial and the invokevirtual instructions is that invokevirtual invokes a method based on the class of the object [7]. The invokespecial instruction has a reference to the class as argument, so the method does not need to be found via an object. The

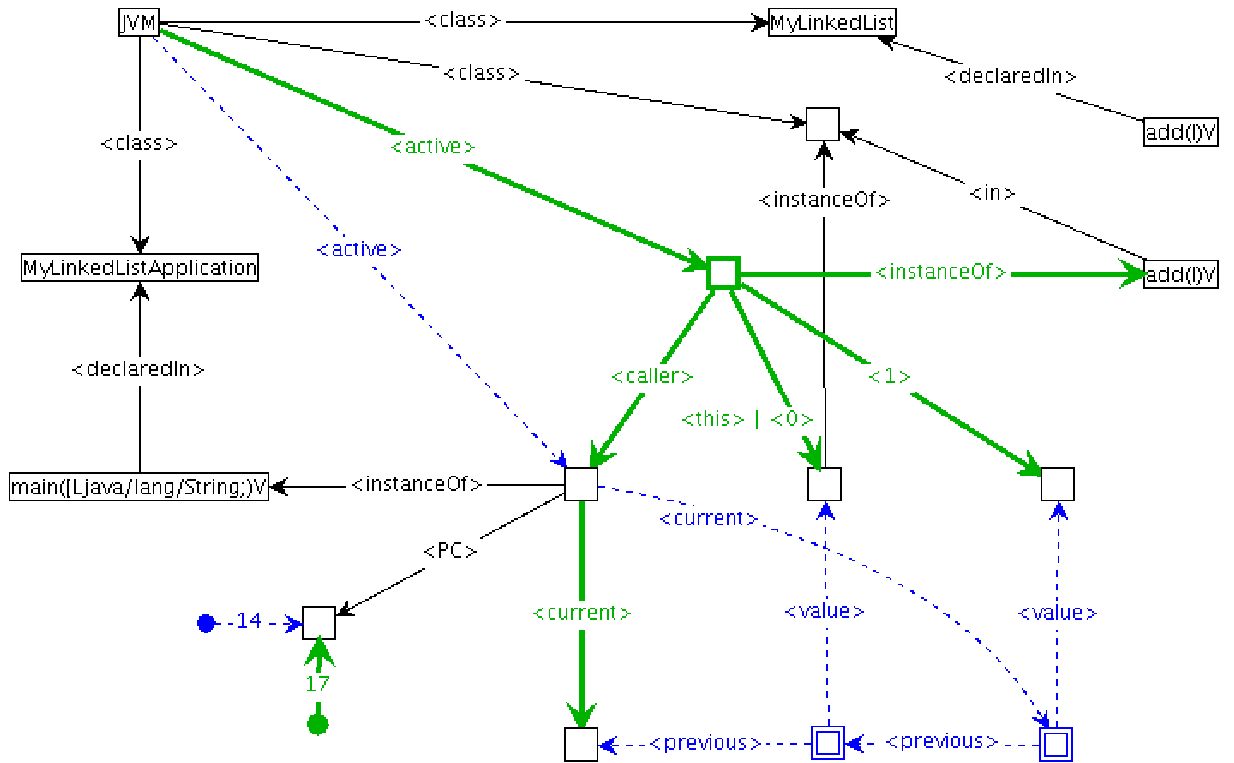

Figure 4.18: Invokevirtual of the add method

invokespecial instruction is used to invoke instance initialisation methods as well as private methods and methods of a superclass of the current class.

The difference in the transformation rule is in how to find the right Method Frame Node to which the $\langle instanceOf \rangle$ needs to go. This can't be found via the class of the object, but must be found via the reference in the instruction. This reference include the class where the method to be invoked is in. In Figure 4.19 the invocation of $\langle init() \rangle$ of class *MyLinkedList* is depicted. The *add* method exists twice in this graph because the method can be declared in a superclass and be overwritten in a subclass.

invokestatic

The difference between the invokestatic and the invokevirtual is that the $\langle this \rangle$ edge goes to the Class Node instead of the Object Node, so there does not need to be an object of that class. Also the $\langle 0 \rangle$ doesn't go to an object, but to the first argument of the method according to the Java Virtual Machine Specification, see [7].

4.4.2 Method execution

A Method is executed after it is invoked. This can be recognized by a method frame node having an $\langle active \rangle$ edge and no $\langle PC \rangle$ edge. The first step of executing a method is adding this $\langle PC \rangle$ edge. This $\langle PC \rangle$ edge is used to order the instructions of a method, see Section 4.1.5.

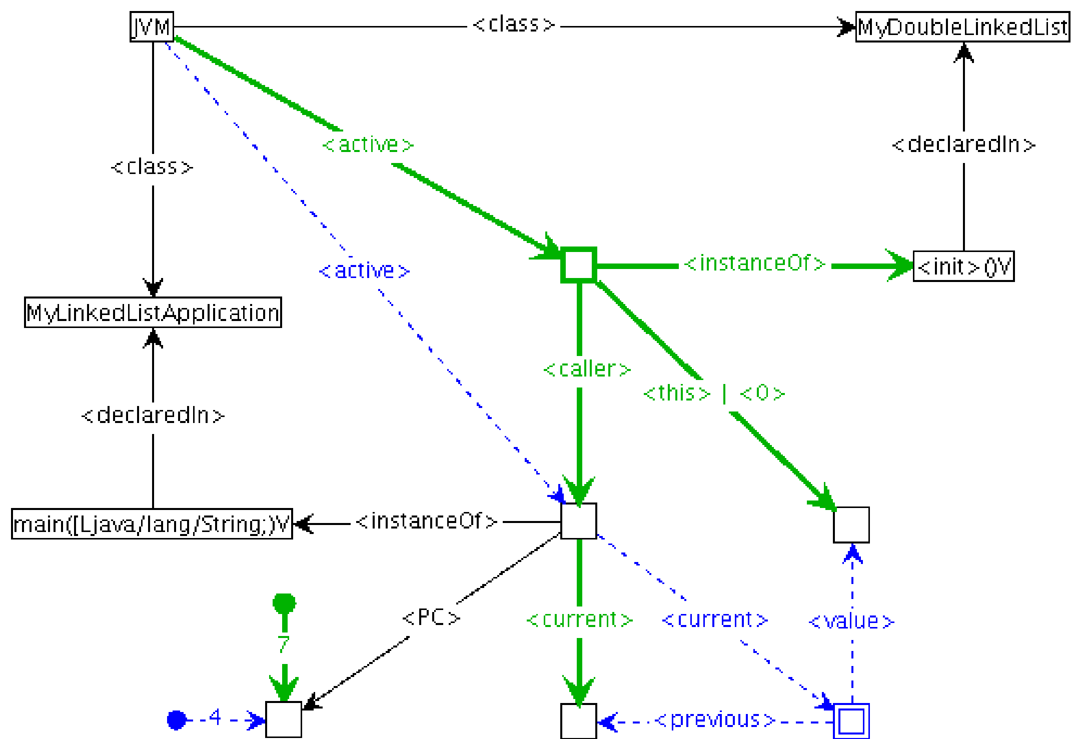

Figure 4.19: Invokespecial of an $\langle init() \rangle$ method

Also a stack has to be created for the Method Frame Node to simulate the stack operations, see Section 4.1.4.

When this is done, the real execution of the method can begin: the execution of the JVM instructions. It will start with the execution of the JVM instruction of which the program counter is zero.

4.4.3 Method return

Every method will return a value, even when the return type is *void*. Because *void* is also represented as a value it can be returned. When a method returns it will delete the $\langle caller \rangle$ and $\langle active \rangle$ edge. It will create an $\langle active \rangle$ edge to the node to calling method to indicate the calling method can continue. The return value is indicated by a $\langle return \rangle$ edge to the caller from the return value.

Furthermore the Method Frame node, stack and program counter of the are deleted.

4.5 Limitations

The implementation of the Graph Virtual Machine has a few limitations at the moment:

- Threads

- Exceptions
- Inner classes
- Arrays

4.5.1 Threads

When there are multiple threads it is possible to execute instructions independently from each other, so for example two methods can be active at the same time, and can be executed on different processors. It is also possible to synchronise between different threads and lock and unlock variables.

In the current implementation threads are not implemented. To implement this the Meta Model has to be changed. This will also result in a lot of changes in the transformation rules. The implementation of threads is planned for the future.

4.5.2 Exceptions

When a program violates the semantic constraints of the Java programming language, the Java virtual machine signals this error to the program as an exception. For example a *null pointer* exception, when a null value is used wrong.

Exceptions are not implemented at the moment this is also planned for the future.

4.5.3 Inner classes

An inner class is a nested class that is not explicitly or implicitly declared static. How to implement inner classes and nested classes is not researched at the moment. But it will not be very hard to implement this, because the nested and inner classes are compiled into separate .class files, so the Java Virtual Machine will probably treat them as normal classes.

4.5.4 Arrays

Arrays of different types are also not implemented yet. At the moment the implementation can work with variables of an array type, as long they are not used. The arrays also have to be implemented in the future.

Chapter 5

Implementation

In this chapter the implementation of the Java byte code to graph grammars translator is described. First the design of the translator is descibed with special attention for the Byte Code Engineering Library. Section 5.2 describes the unit testing and system testing of the translator. In Section 5.3 the usage of the translator is explained. The last section is about the limitations of the translator.

5.1 Design of the translator

To translate the binary byte code there are two problems to be solved:

- Interpret the byte code
- Translate the byte code instructions into graph production rules

The first problem is a very general problem. To read and interpret Java byte code there are already different solutions. A well-known solution to interpret, analyze and even manipulate Java byte code is the Byte Code Engineering Library (BCEL)[1], see Section 5.1.1. We choose BCEL because it is widely used and is freely available under the terms of the Apache Licence. A famous application of BCEL is in AspectJ [2].

The second problem is more complicated. By using the knowledge in Chapter 4 we know how to represent a program in graph grammars. By using this knowledge and the Java Virtual Machine Specification [7] it is possible to create production rules for each JVM instruction. These production rules needs to be adapted to each different situation and therefore are called templates, see 5.1.3.

[1] The distribution is available at `http://jakarta.apache.org/bcel/`, including several code examples and javadoc manuals.
[2] Available at `http://eclipse.org/aspectj/`

5.1.1 Byte Code Engineering Library

The use of the Byte Code Engineering Library (BCEL) is very important for the design of the translator. This library is a toolkit for static analysis and dynamic creation or transformation of Java class files. With this library you can implement the desired features on a high level of abstraction without handling all the internal details of the Java class file format and thus re-inventing the wheel.

The API of BCEL consists of three parts[2]:

1. A package that contains classes that describe ”static” constraints of class files, i.e., reflects the class file format and is not intended for byte code modifications. The classes may be used to read and write class files from or to a file. This is useful especially for analyzing Java classes without having the source files at hand. The main data structure is called *JavaClass* which contains methods, fields, etc..

2. A package to dynamically generate or modify *JavaClass* or *Method* objects. It may be used to insert analysis code, to strip unnecessary information from class files, or to implement the code generator back-end of a Java compiler.

3. Various code examples and utilities like a class file viewer, a tool to convert class files into HTML, and a converter from class files to the Jasmin assembly language.

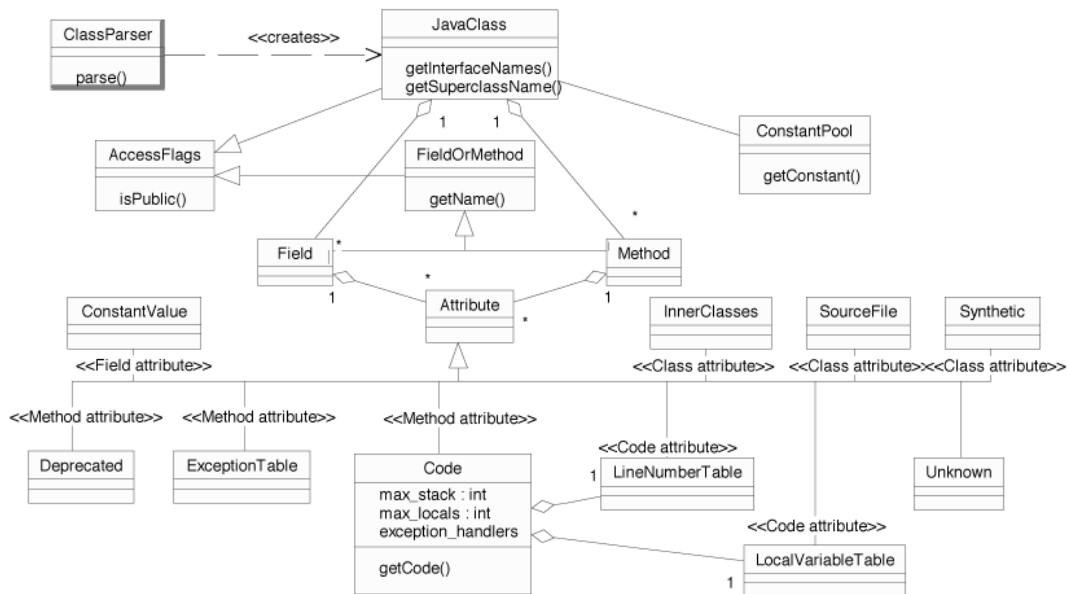

Figure 5.1: UML diagram of the BCEL API [2]

Only the first part of the API is used for the translator. All of the binary components and data structures declared in the JVM specification [7] are mapped to classes. In Figure 5.1 the UML diagram of the *JavaClass* structure is given. The *JavaClass* data structure is in most cases created by a *ClassParser* object. This *ClassParser* object is capable of parsing binary class

files. A JavaClass object basically consists of fields, methods, symbolic references to the super class and to the implemented interfaces.

This JavaClass structure can by analysed by using a visitor design pattern [4]. With the visitor pattern we can define new operations without changing the classes of the elements on which it operates. The object structure that we are visiting, the Java class, consists of different classes of objects like: JavaClass, Field, and Method. We want to perform different operations on these objects, the operation to be performed depend on the class it operates on. This is why a visitor pattern is suitable for this situation.

With a visitor pattern each element of the object structure implements an *accept(Visitor)* method. This *accept* method will invoke a method of the visitor named visitClassname for example visitField, as argument it passes its own type, for example visitField(Field). This way the right method of the visitor is invoked. The visitor can also use methods of the class that it is visiting. In Figure 5.2 a sequence diagram is given from the visitor pattern.

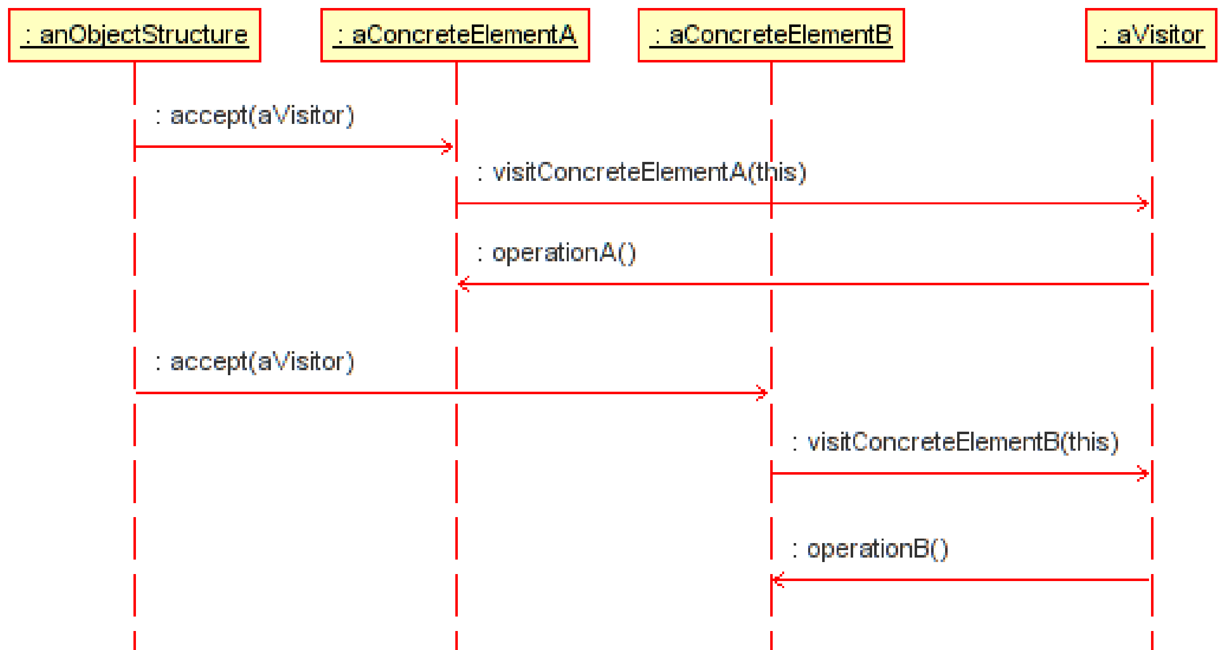

Figure 5.2: Sequence diagram of visitor pattern

Because a visitor pattern is used, the actual design of the translator is simple; it is an visitor for the elements of the JavaClass. The JavaClass structure is provided by the BCEL library. The visitor must have methods for all the different classes in the JavaClass structure. The BCEL library provides a Visitor interface; in this interface all methods to be implemented by a visitor are declared. Also there is an empty implementation of this Visitor interface, so subclasses only need to implement the methods they need.

The exact visitor pattern implemented is a little bit different than described above, because BCEL provides a *DescendingVisitor*. This visitor traverses a JavaClass with another Visitor object 'piggy-backed' that is applied to all components of a JavaClass object. This class supplies the traversal strategy, and other classes can use this. This way the own developed visitor

does not have to traverse the JavaClass object structure by itself and it becomes simple to implement the visitor pattern, only the required methods needs to be implemented. In Figure 5.3 a part of the sequence diagram of the JBCTranslator and the ClassfileVisitor 'piggy-backed' in the *DescendingVisitor*.

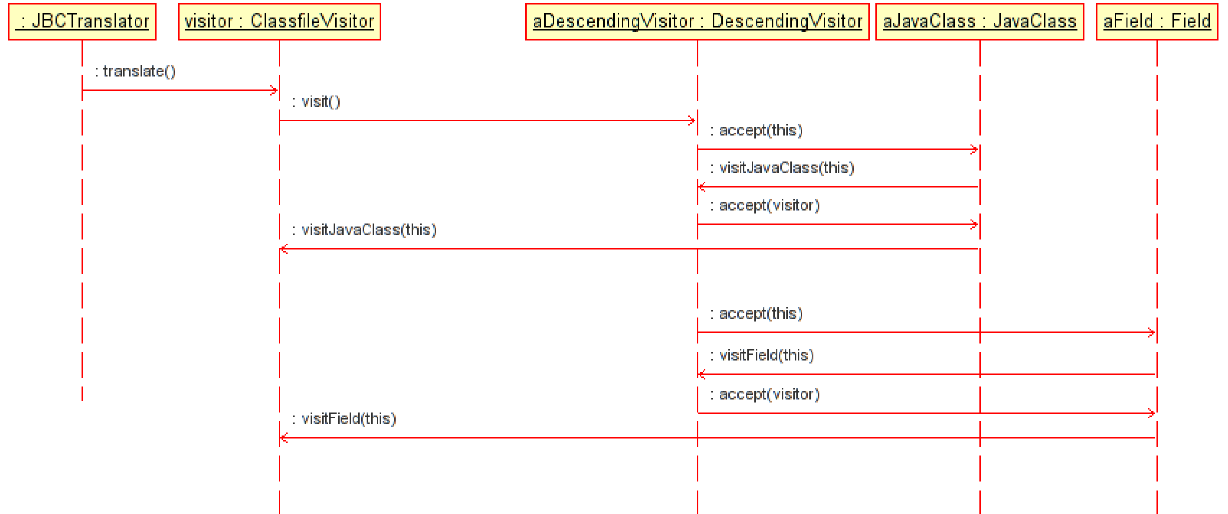

Figure 5.3: Part of the sequence diagram of translator

5.1.2 Classes of the translator

The most important class in the translator will be the class that visits the JavaClass structure. This class will provide all the methods needed for translating the byte code into graph grammars. An overview og the classes that needs to be implemented:

- *JBCTranslator*: The main class, this will read the command line parameters, parse the given classes by using BCEL and start the translation of them. JBCTranslator stands for: Java Byte Code Translator
- *ClassfileVisitor*: The actual translator, this will visit the JavaClass structure.
- *ReplaceContainer*: A container for all variables to be replaced by the *GraphVarReplacer* class. Simple structure containing information about which variable in a template production rule will be replaced by what string, see Section 5.1.3.
- *GraphVarReplacer*: A class that replaces all variable labels in a graph, and saves the graph at a specified location. The information about which variable in a graph should be replaced by which string in a *ReplaceContainer*.
- *Processed*: This is a class needed for building the meta graph. This meta graph is used for testing purposes, see Section 5.2. In this class all labels are stored which are used during the translation.

In Figure 5.4 the class diagram of the translator is given, also a small part of the classes provided by the BCEL library is given.

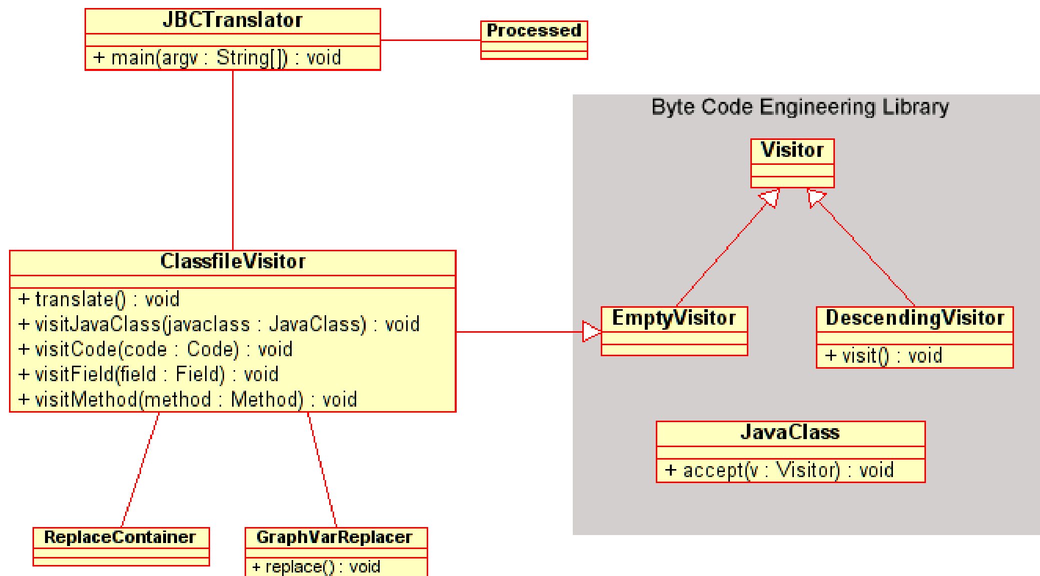

Figure 5.4: Classdiagram

5.1.3 Templates

For each Java byte code instruction it is specified in [7] what the instruction does. So for each instruction it is known what it should do, so this can be put into a production rule. The only difference between each application of the same rule is the exact labels of nodes and edges. For this reason for all production rules, which correspond to byte code instructions, a template is made. A template consists of all common nodes and labels of an instruction. All variable labels are in the template as variables, an example is a classname which is different in each application of the rule. These variable labels are replaced when the real production rule is made. In Figure 5.5 is an example of a template. In this example the template of the *dup* instruction is depicted. The labels starting with a $ are variables and they need to be replaced by the right values, for example the program counter for this instruction.

In Figure 5.6 is a possible application of the template. In this figure all variable labels are replaced with real values.

The replacing of the variables in a graph is done by the class *GraphVarReplace*. The knowledge of which variable needs to be replaced by what label is in the class *ReplaceContainer*. The replacer uses this knowledge to relabel the labels in the graph into the right labels. The relabeling of labels is a standard function of the GROOVE toolset.

The replace graph also saves the generated production rule to the right location. The functionality for loading and saving of production rules is also provided by GROOVE.

5.1.4 Production rules for templates

It is not always possible to solve the problem of creating the right production rules by only using variable labels. Sometimes it is also needed to create extra nodes and edges depending

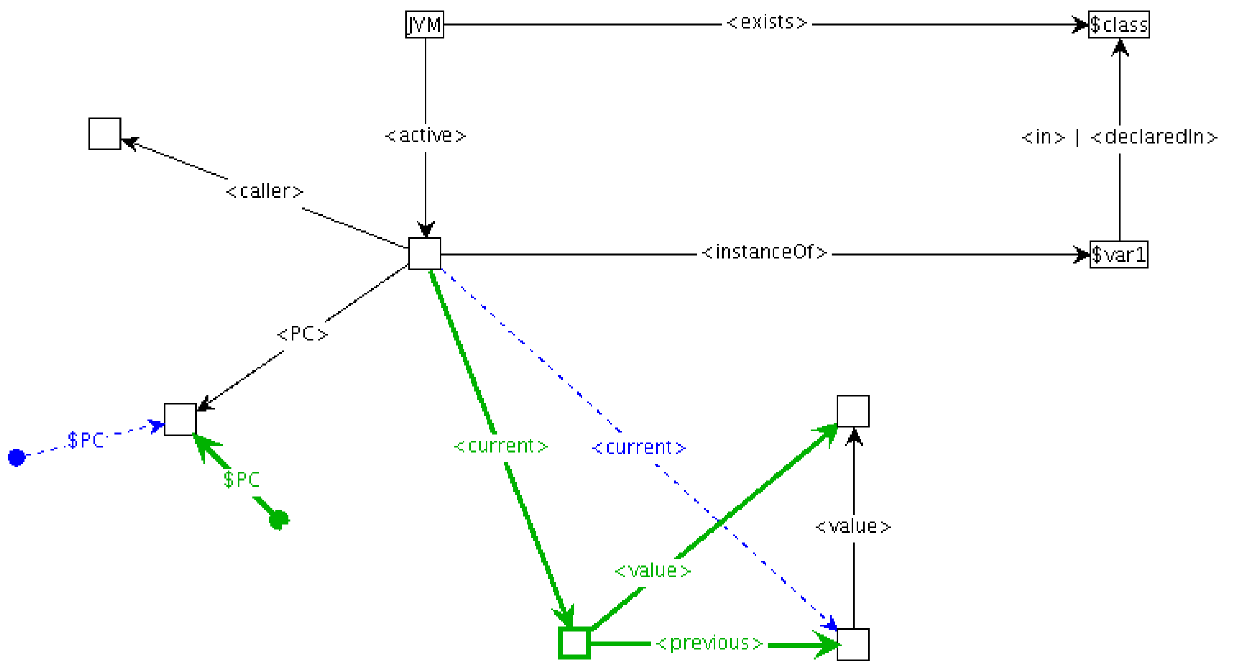

Figure 5.5: Template of the *dup* instruction

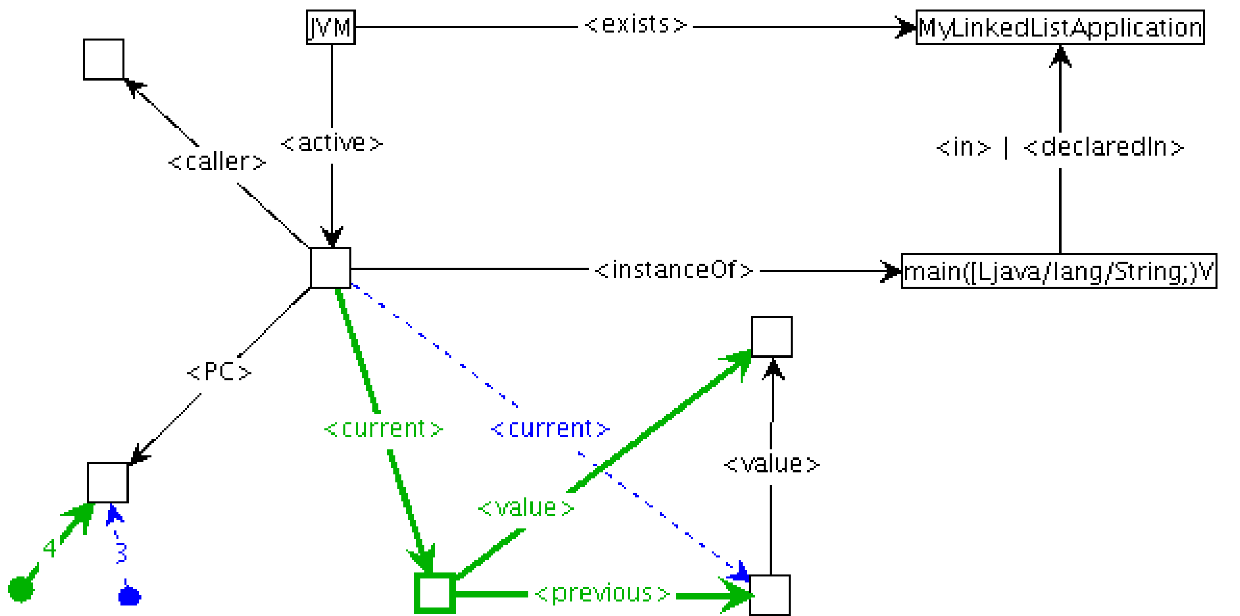

Figure 5.6: Application of the *dup* template

on the situation. For example the invocation of a method can be different; the number of arguments that needs to be passed to the method is variable. For this reason, also the templates need to be transformed.

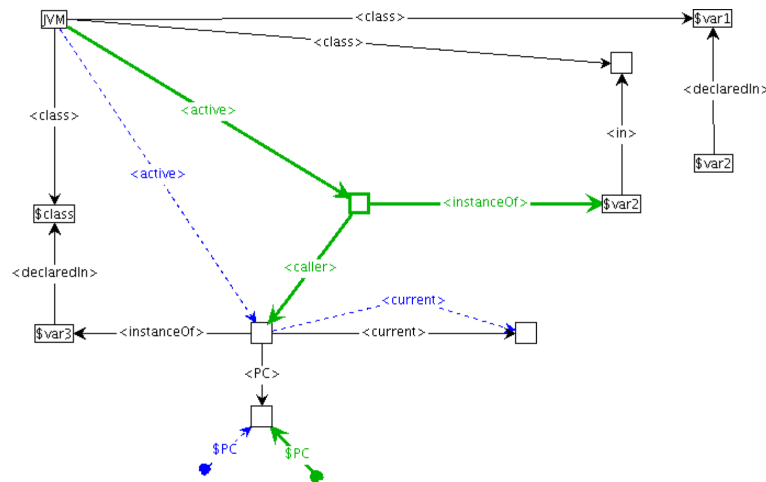

Figure 5.7: Template of *invoke_virtual*

In Figure 5.7 the template for the instruction *invoke_virtual* is given, see Section 4.4.1 for more information about the *invoke_virtual* instruction. This instruction can have a variable number of arguments, so this template needs to be transformed to have the right number of arguments for each situation. The transformation of the production rules themselves can also be done by applying a production rule. This is because a production rule itself is also a graph in GROOVE. In Figure 5.8 the template *invoke_virtual* is given in the GROOVE input format. By using this format it is possible to define a production rule to transform the template, because also matching on the creation and deletion of nodes and edges can be machted: the labels *new:* and *del:* can be used as it are normal labels.

In Figure 5.9 the production rule for the transformation of the template is given in the GROOVE input format. By applying this production rule a new argument is added to the template. In Figure 5.10 the same rule is given as a production rule.

In Figure 5.11 the result of applying the production rule to the template is given. After applying the necessary production rules to the template, the variable labels needs to be replaced by the right labels.

5.2 Testing

Testing of the translator is done in two ways: unit testing and system testing. With unit testing the functionality of classes is tested separately. With system testing the whole system is tested as a whole.

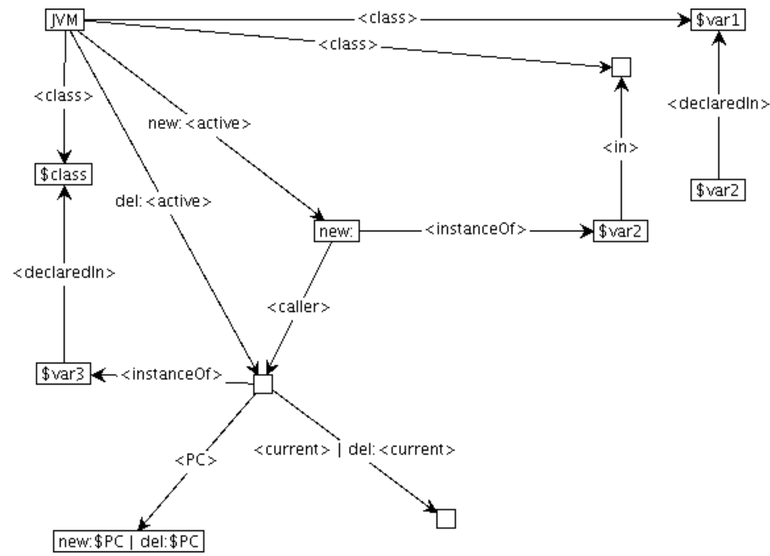

Figure 5.8: Template of *invoke_virtual* in GROOVE input format

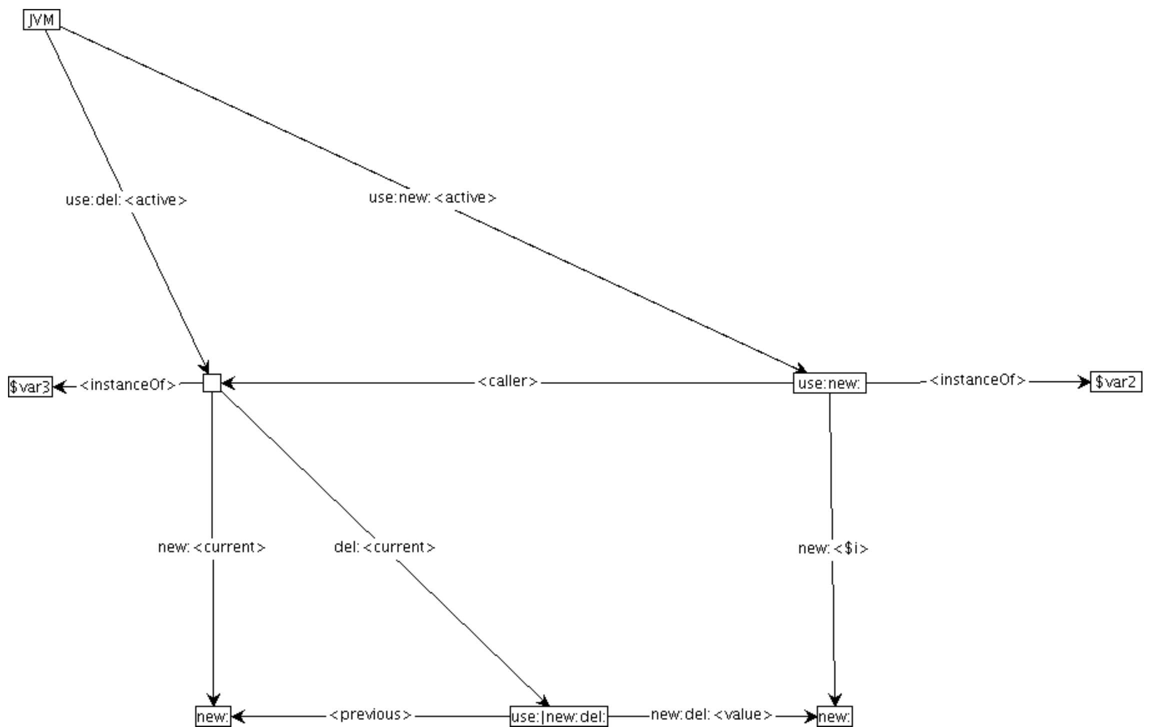

Figure 5.9: Production rule to be applied on the template in the GROOVE input format

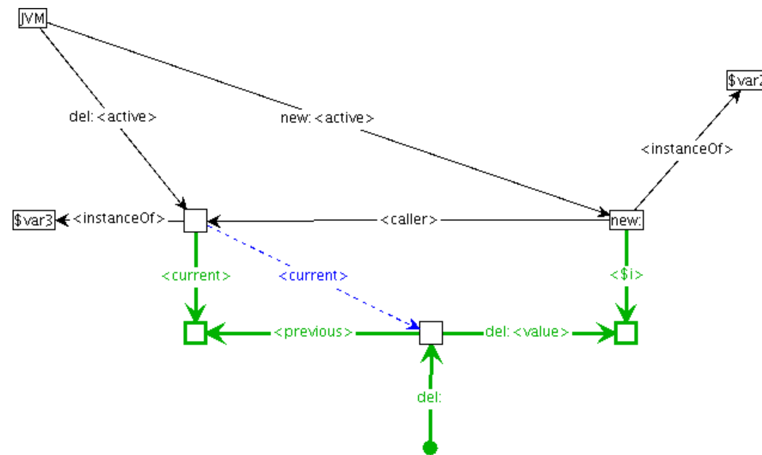

Figure 5.10: Production rule to be applied on the template

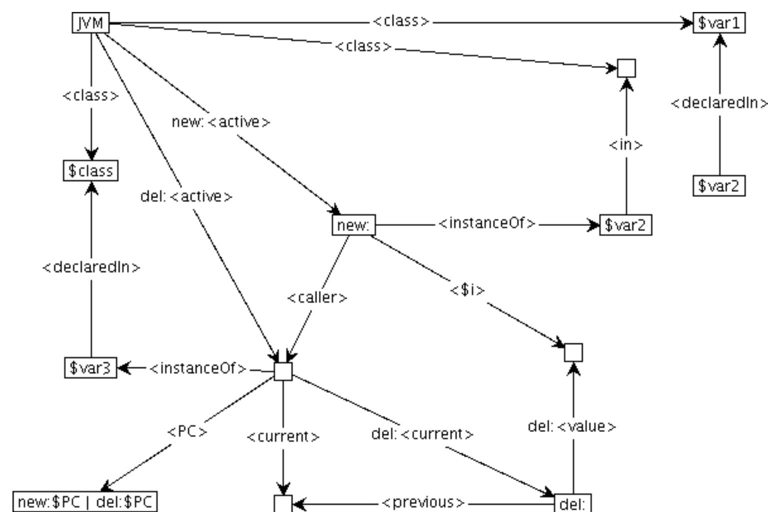

Figure 5.11: Result of application of the production rule on the template

5.2.1 Unit testing

Unit testing is done by using JUnit [3]. With JUnit separate test classes are made to test separate functionality of the software.

The class that is fully tested by using JUnit is the *GraphVarReplace* class. This is a very important class because it replaces the variable labels in graphs by the right values for each situation. This class also saves the rules to the right location. This class is tested by 13 test cases testing the following:

- Test if needed directories for the output are created.

[3] Available at `http://www.junit.org` under the IBM's Common Public License Version 1.0

- Test if all different variable labels are relabeled correctly.
- Test if rules are saved correctly and on the right location.

The other classes are at the moment not tested by using unit tests. The *ClassVisitor* is tested by hand. Because the *ClassVisitor* is implemented incrementally, each time one JVM instruction more, it was possible to look if the produced production rule is the desired one. The most error prone in the translation are errors introduced by building the template files by hand.

In the future also the *ClassVisitor* class needs to be tested by unit testing. This is a lot of work; the graph before a JVM instruction and the graph after the instruction must be made by hand. When these two states are available it is possible to see if the instruction transforms the start state into the desired end state.

5.2.2 System testing

System testing consists of two parts: checking if the program runs and checking if the output is right. The first part is tested by using the translator for translating the classes of the linked list example. The translator translates the class files and put the output in the desired directories. Checking if the output of the system makes any sense is done automatically. This is done by checking if all states of the graph transition system are correct. A state is correct if it complies to the Meta model of Section 4.2. A state complies to the Meta model if it has a match to the Meta model. Before this can be checked, a number of labels need to be added to the Meta model because the model contains not all labels. For example the label *identifiers* in the Meta model is a set of labels, it contains of all identifiers used in the system. So, all identifier names need to be inserted into the Meta model. Also names of classes and methods are not in the Meta model. During the translation all the labels used are collected, and after the translation the Meta model for the system is created with all the labels in it.

This way most errors in production rules can be found, but not all errors. If the created system complies with the Meta model this does not mean it is exactly the right system. For this reason also checking by hand is done. This is done by manually applying production rules to the initial state and look if the new state is as expected. Also after implementing the translation of a JVM instruction the generated production rules are checked. Because this is done by hand it is very time consuming and error-prone.

5.3 Usage

The translator is located in the GROOVE project in the package *groove.jvm*. The class with the main method is *JBCTranslator*. The usage of the JBCTransaltor is as follows:

java groove.jvm.JBCTranslator *input_dir output_dir classes*

The different options are:

java The Java virtual machine executable

groove.jvm.JBCTranslator The class with the main method, the Java Byte Code Translator

input_dir The input directory. In this directory the *.class* files must be located. If files are in different locations, you will have to give the full location with the class in the option *classes*.

output_dir The output directory. In this directory the created files will be placed. This directory will be created if it does not exist. This is the directory you can load in to the simulator after the translation.

classes These are the classes to translate. The classnames can be given or the filenames of the class files with full location. The last class in this list must be the class with a main method. The main method of this (the last) class will be invoked by the JVM.

5.4 Limitations

The limitations of the current implementation are apart from the limitation mentioned in Section 4.5 that not all JVM instructions are implemented. The most important instructions are implemented at the moment. For example, instructions for other types than *int* and *boolean* are not implemented right now. In Appendix D a complete overview of implemented JVM instructions is given.

Chapter 6

Conclusions and recommendations

The outcome of this research is that automatic translation of Java byte code into a system of graph production rules, graph grammars, is possible. Also a working translator is written. With the resulting system of graph production rules it is possible to analyse the run-time behaviour of a program. Right now it is possible to calculate all reachable states of a program. The difference with other model checkers is that with this graph grammar we are also able to analyse the dynamic behaviour of a program, for example the dynamic (de)allocation, method calls and returns.

To be able to do the translation from Java byte code into graph grammars a Meta model is created. This Model describes how to represent every run-time state of a Java program in a graph. In this model different kinds of nodes and edges are defined. The edges between the nodes represent the relation between the nodes, for example a certain object is an instance of a class. Different design choices are made during the design of this model. These choices are made with the Java Virtual Machine Specification [7] in mind, so the simulation is as close as possible to how a program runs in the Java Virtual Machine.

Using this representation a translator is build that is capable of translating Java byte code into graph grammars. Not all JVM instructions are implemented yet, so only very simple Java programs can be translated. The example used in this report can reach 111 different states. This state space can become very huge when a complicated program is being analysed. In the future a way must be found to reduce this state space to be able to analyze complete programs.

For each implemented JVM instruction a template is made. This template is used to build the production rule for each situation by inserting the right labels. Some instructions have variable parts, like the number of arguments in a method call. This is solved by having production rules to transform the templates. However, these templates need to be build by hand which is error-prone. This process needs to be done more systematically: for each production rule a specification must be written, like the Java Virtual Machine Specification. The source graph and the desired target graph for each production rule must be specified. When this is done also automatic testing of each production rule will be possible with Unit testing.

To reduce the number of possible states, the production rules now predefine an order of loading, linking and initialisation of classes. Without this ordering the simulator will calculate all possible orderings of this process. The disadvantage of our solution is that the production rules are very complex.

In the future all missing JVM instructions and the missing Java concepts need to be implemented like threads, exceptions and arrays. Also more research needs to be done in how to analyse the generated state space, so that bugs in the software can really be found. When this is done, this translator can contribute to finding more bugs in software.

Bibliography

[1] Bandera - java model checker. Project URL: http://bandera.projects.cis.ksu.edu.

[2] M. Dahm. Byte code engineering library (bcel) description and usage manual, October 2001. Available at http://jakarta.apache.org/bcel/manual.html.

[3] 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.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, NY, 1995.

[5] A. Habel, R. Heckel, and G. Taentzer. Graph grammars with negative application conditions. *Fundamenta Informaticae*, 26(3/4):287-313, 1996.

[6] Java™native interface. Available at http://java.sun.com/products/jdk/1.2/docs/guide/jni/.

[7] T. Lindholm and F. Yellin. *The Java virtual machine specification*. Addison Wesley, second edition, 1999.

[8] M. Löwe. Algebraic approach to single-pushout graph transformation. *Theoretical Computer Science*, 109(1-2):181-224, 1993.

[9] Java pathfinder - a formal methods tool for java. Project URL: http://ase.arc.nasa.gov/people/havelund/jpf.html.

[10] A. Rensink. Graphs for object-oriented verification: A tool set for graph grammar simulation, 2003. Available at http://www.cs.utwente.nl/~groove.

[11] A. Rensink. Towards model checking graph grammars. In M. Leuschel, S. Gruner, and S. Lo Presti, editors, *Proceedings of the* 3rd *Workshop on Automated Verification of Critical Systems*, Technical Report DSSE–TR–2003–2, pages 150–160. University of Southampton, 2003.

[12] A.C. Lozano Rodríguez. A representation of java programs as partial graph morphisms. Master's thesis, Ecole des Mines de Nantes and University of Twente, 2003.

Appendix A

Linked list example source code

```java
public class MyLinkedListApplication{
    private static MyLinkedList linkedList;

    public static void main(String[] args){
        linkedList = new MyDoubleLinkedList();
        linkedList.add(1);
        linkedList.add(2);
        linkedList.add(3);
        int value = linkedList.getValue();
    }
}
```

Listing A.1: MyLinkedListApplication.java

```java
class MyLinkedList{
    public MyLinkedListNode first;


    public void add(int v){
        MyLinkedListNode newNode = new MyLinkedListNode(v);
        newNode.next = first;
        first = newNode;

    }

    public int getValue(){
        return first.getValue();
    }
}
```

Listing A.2: MyLinkedList.java

```java
class MyLinkedListNode
{
    private int value;
    public MyLinkedListNode next;


    MyLinkedListNode(int v)
    {
        value = v;
        next = null;
    }

    public int getValue(){
        return value;
    }

}
```

Listing A.3: MyLinkedListNode.java

```java
class MyDoubleLinkedList extends MyLinkedList{


    public void add(int v){

        MyDoubleLinkedListNode newNode = new MyDoubleLinkedListNode(
            v);

        newNode.next = first;
        if (first!=null){
            ((MyDoubleLinkedListNode) first).prev = newNode;
        }
        first = newNode;
    }

}
```

Listing A.4: MyDoubleLinkedList.java

```
class MyDoubleLinkedListNode extends MyLinkedListNode {

    public MyDoubleLinkedListNode prev;

    MyDoubleLinkedListNode(int v){
        super(v);
        prev = null;
    }
}
```

Listing A.5: MyDoubleLinkedListNode.java

Appendix B

Linked list example byte code

```
public class MyLinkedListApplication extends java.lang.Object
filename        MyLinkedListApplication
compiled from       MyLinkedListApplication.java
compiler version    46.0
access flags        33
constant pool       32 entries
ACC_SUPER flag      true

Attribute(s):
    SourceFile(MyLinkedListApplication.java)

1 fields:
    private static MyLinkedList linkedList

2 methods:
    public void <init>()
    public static void main(String[] arg0)

public void <init>()
Code(max_stack = 1, max_locals = 1, code_length = 5)
0:    aload_0
1:    invokespecial java.lang.Object.<init> ()V (1)
4:    return

Attribute(s) =
LineNumber(0, 1)

public static void main(String[] arg0)
Code(max_stack = 2, max_locals = 2, code_length = 39)
0:    new       <MyDoubleLinkedList> (2)
3:    dup
4:    invokespecial MyDoubleLinkedList.<init> ()V (3)
7:    putstatic     MyLinkedListApplication.linkedList LMyLinkedList
    ; (4)
10:   getstatic     MyLinkedListApplication.linkedList LMyLinkedList
    ; (4)
```

```
35 13:   iconst_1
36 14:   invokevirtual MyLinkedList.add (I)V (5)
37 17:   getstatic     MyLinkedListApplication.linkedList LMyLinkedList
       ; (4)
38 20:   iconst_2
39 21:   invokevirtual MyLinkedList.add (I)V (5)
40 24:   getstatic     MyLinkedListApplication.linkedList LMyLinkedList
       ; (4)
41 27:   iconst_3
42 28:   invokevirtual MyLinkedList.add (I)V (5)
43 31:   getstatic     MyLinkedListApplication.linkedList LMyLinkedList
       ; (4)
44 34:   invokevirtual MyLinkedList.getValue ()I (6)
45 37:   istore_1
46 38:   return
47
48 Attribute(s) =
49 LineNumber(0, 5), LineNumber(10, 6), LineNumber(17, 7), LineNumber
       (24, 8),
50 LineNumber(31, 9), LineNumber(38, 10)
51
52 Ooops
```

Listing B.1: MyLinkedListApplication.bytecode

```
class MyLinkedList extends java.lang.Object
filename        MyLinkedList
compiled from       MyLinkedList.java
compiler version    46.0
access flags        32
constant pool       30 entries
ACC_SUPER flag      true

Attribute(s):
    SourceFile(MyLinkedList.java)

1 fields:
    public MyLinkedListNode first

3 methods:
    void <init>()
    public void add(int arg1)
    public int getValue()

void <init>()
Code(max_stack = 1, max_locals = 1, code_length = 5)
0:    aload_0
1:    invokespecial java.lang.Object.<init> ()V (1)
4:    return

Attribute(s) =
LineNumber(0, 1)

public void add(int arg1)
Code(max_stack = 3, max_locals = 3, code_length = 23)
0:    new       <MyLinkedListNode> (2)
3:    dup
4:    iload_1
5:    invokespecial MyLinkedListNode.<init> (I)V (3)
8:    astore_2
9:    aload_2
10:   aload_0
11:   getfield      MyLinkedList.first LMyLinkedListNode; (4)
14:   putfield      MyLinkedListNode.next LMyLinkedListNode; (5)
17:   aload_0
18:   aload_2
19:   putfield      MyLinkedList.first LMyLinkedListNode; (4)
22:   return

Attribute(s) =
LineNumber(0, 6), LineNumber(9, 7), LineNumber(17, 8), LineNumber
   (22, 10)


public int getValue()
Code(max_stack = 1, max_locals = 1, code_length = 8)
0:    aload_0
```

```
52 1:    getfield      MyLinkedList.first LMyLinkedListNode; (4)
53 4:    invokevirtual MyLinkedListNode.getValue ()I (6)
54 7:    ireturn
55
56 Attribute(s) =
57 LineNumber(0, 13)
58
59 Ooops
```

Listing B.2: MyLinkedList.bytecode

```
class MyLinkedListNode extends java.lang.Object
filename        MyLinkedListNode
compiled from       MyLinkedListNode.java
compiler version    46.0
access flags        32
constant pool       24 entries
ACC_SUPER flag      true

Attribute(s):
    SourceFile(MyLinkedListNode.java)

2 fields:
    private int value
    public MyLinkedListNode next

2 methods:
    void <init>(int arg1)
    public int getValue()

void <init>(int arg1)
Code(max_stack = 2, max_locals = 2, code_length = 15)
0:    aload_0
1:    invokespecial java.lang.Object.<init> ()V (1)
4:    aload_0
5:    iload_1
6:    putfield      MyLinkedListNode.value I (2)
9:    aload_0
10:   aconst_null
11:   putfield      MyLinkedListNode.next LMyLinkedListNode; (3)
14:   return

Attribute(s) =
LineNumber(0, 8), LineNumber(4, 9), LineNumber(9, 10), LineNumber
    (14, 11)


public int getValue()
Code(max_stack = 1, max_locals = 1, code_length = 5)
0:    aload_0
1:    getfield      MyLinkedListNode.value I (2)
4:    ireturn

Attribute(s) =
LineNumber(0, 14)

Ooops
```

Listing B.3: MyLinkedListNode.bytecode

```
class MyDoubleLinkedList extends MyLinkedList
filename          MyDoubleLinkedList
compiled from         MyDoubleLinkedList.java
compiler version      46.0
access flags          32
constant pool         33 entries
ACC_SUPER flag        true

Attribute(s):
    SourceFile(MyDoubleLinkedList.java)

2 methods:
    void <init>()
    public void add(int arg1)

void <init>()
Code(max_stack = 1, max_locals = 1, code_length = 5)
0:    aload_0
1:    invokespecial MyLinkedList.<init> ()V (1)
4:    return

Attribute(s) =
LineNumber(0, 1)

public void add(int arg1)
Code(max_stack = 3, max_locals = 3, code_length = 41)
0:    new       <MyDoubleLinkedListNode> (2)
3:    dup
4:    iload_1
5:    invokespecial MyDoubleLinkedListNode.<init> (I)V (3)
8:    astore_2
9:    aload_2
10:   aload_0
11:   getfield      MyDoubleLinkedList.first LMyLinkedListNode; (4)
14:   putfield      MyDoubleLinkedListNode.next LMyLinkedListNode
   ; (5)
17:   aload_0
18:   getfield      MyDoubleLinkedList.first LMyLinkedListNode; (6)
21:   ifnull        #35
24:   aload_0
25:   getfield      MyDoubleLinkedList.first LMyLinkedListNode; (7)
28:   checkcast     <MyDoubleLinkedListNode> (2)
31:   aload_2
32:   putfield      MyDoubleLinkedListNode.prev
   LMyDoubleLinkedListNode; (8)
35:   aload_0
36:   aload_2
37:   putfield      MyDoubleLinkedList.first LMyLinkedListNode; (9)
40:   return

Attribute(s) =
```

```
50 LineNumber(0, 6), LineNumber(9, 8), LineNumber(17, 9), LineNumber
       (24, 10),
51 LineNumber(35, 12), LineNumber(40, 13)
52
53 Ooops
```

Listing B.4: MyDoubleLinkedList.bytecode

```
class MyDoubleLinkedListNode extends MyLinkedListNode
filename        MyDoubleLinkedListNode
compiled from       MyDoubleLinkedListNode.java
compiler version    46.0
access flags        32
constant pool       17 entries
ACC_SUPER flag      true

Attribute(s):
    SourceFile(MyDoubleLinkedListNode.java)

1 fields:
    public MyDoubleLinkedListNode prev

1 methods:
    void <init>(int arg1)

void <init>(int arg1)
Code(max_stack = 2, max_locals = 2, code_length = 11)
0:    aload_0
1:    iload_1
2:    invokespecial MyLinkedListNode.<init> (I)V (1)
5:    aload_0
6:    aconst_null
7:    putfield      MyDoubleLinkedListNode.prev
   LMyDoubleLinkedListNode; (2)
10:   return

Attribute(s) =
LineNumber(0, 6), LineNumber(5, 7), LineNumber(10, 8)

Ooops
```

Listing B.5: MyDoubleLinkedListNode.bytecode

Appendix C

Description of all Nodes and Edges

C.1 JVM Node

Description To simulate the complete behaviour of application also the JVM needs to be simulated. For example it keeps track of all loaded classes and calls the main method of the first loaded class.

Representation Represented by a node with a self edge called JVM.

Outgoing edges

- $\langle exists \rangle$ to a Class Node to indicate the class is known by the JVM
- $\langle load \rangle$ to a Class Node to indicate the class needs to be loaded
- $\langle loading \rangle$ to a Class Node to indicate the class is being loaded
- $\langle link \rangle$ to a Class Node to indicate the class needs to be linked
- $\langle init \rangle$ to a Class Node to indicate the class needs to be initialized
- $\langle initing \rangle$ to a Class Node to indicate the class is being initialized
- $\langle class \rangle$ to a Class Nodes to indicate it is loaded, linked and initialized
- $\langle 1 \rangle$ to the first argument, the Class Node with the main method that needs to be called
- $\langle 2 \rangle$ to the second argument, the argument that is passed tot the main method
- $\langle active \rangle$ to the active method or the JVM Node itself
- $\langle PC \rangle$ edge to an Instruction Order Node, used for loading classes in the right order

Incoming edges

- $\langle caller \rangle$ from the main method it is the caller of
- $\langle return \rangle$ from the return value of a called method

C.2 Interface Node

Description An Interface Node has only constants fields and abstract methods. It has no implementation; the implementation will be in the implementing class.

Representation Interfaces are represented by a node with a self edge. The fully qualified name of the interface is the label of the self edge.

Outgoing edges

- $\langle super \rangle$ edge to a super interface

Incoming edges

- $\langle in \rangle$ edges from Method Signatures
- $\langle implements \rangle$ from classes that implement this interface
- $\langle super \rangle$ from a sub-interface

C.3 Class Node

Description Class Nodes represent classes and interfaces in a running Java application. The primitive Java types (boolean, byte, char, short, int, long, float, and double), and the special type null are also represented as Class Nodes.

Representation A Class Node is represented by a node with a self edge with its (unique) identifier as label.

Outgoing edges

- $\langle super \rangle$ edge to its super class
- $\langle implements \rangle$ edges to Interface Nodes this class is implementing
- edges to its static fields (Object Nodes) with identifier as labels

Incoming edges

- $\langle super \rangle$ edges from its sub classes
- $\langle instanceOf \rangle$ edges from instantiated Object Nodes
- $\langle this \rangle$ edges from its static methods
- $\langle in \rangle$ edges from the Method Signature Nodes (also from its inherited methods)
- $\langle declaredIn \rangle$ edge from the Method Signatures Nodes that are declared in this class
- $\langle exists \rangle$ from the JVM to indicate the class is known by the JVM
- $\langle load \rangle$ from the JVM to indicate the class needs to be loaded
- $\langle loading \rangle$ from the JVM to indicate the class is being loaded
- $\langle link \rangle$ from the JVM to indicate the class needs to be linked
- $\langle init \rangle$ from the JVM to indicate the class needs to be initialized
- $\langle initing \rangle$ from the JVM to indicate the class is being initialized
- $\langle class \rangle$ from the JVM to indicate it is loaded, linked and initialized

C.4 Object Node

Description An Object Node represents an instance of a Class Node in a running Java application.

Representation An Object Node has no self edge. Except when it is an instance of a primitive type, it has a self edge representing its value.

Outgoing edges

- $\langle instanceOf \rangle$ edge to the Class Node it is an instance of
- $\langle super \rangle$ edge to an instance of its super class
- $\langle return \rangle$ to a caller, representing this node as a return value

Incoming edges

- $\langle this \rangle$ edge from its methods
- edge with the identifier as label to a Method Frame Node
- $\langle super \rangle$ edge from an instance of its sub-class

C.5 Method Signature Node

Description A Method Signature represents the signature of a method.

Representation A Method Signature Node is represented by a node with a self edge with the method name followed by it signature as label.

Outgoing edges

- $\langle in \rangle$ edge to the class where the method is in
- $\langle declaredIn \rangle$ edge to the class where the method is declared in

Incoming edges

- $\langle instanceOf \rangle$ edges from Method Frame Nodes

C.6 Method Frame Node

Description A Method Frame Node represents the frame of a running method.

Representation A Method Frame Node is represented by a node without a self edge.

Outgoing edges

- $\langle this \rangle$ edge to the Class Node if it is a static method and otherwise to the corresponding object
- $\langle instanceOf \rangle$ edge to the Method Signature Node it belongs to
- $\langle caller \rangle$ edge to the Method or JVM that called this Method
- edges to its local fields with identifiers as labels
- $\langle PC \rangle$ edge to a Instruction Order Node

Incoming edges

- $\langle caller \rangle$ edge from any Method that is called by this Method
- $\langle active \rangle$ from the JVM Node to indicate this Method Frame Node is active

C.7 Instruction Order Node

Description An Instruction Order Node is used to model the sequence of instructions determined by the source code inside a Method Frame Node.

Representation An Instruction Order Node is represented by a node with a self edge with a number as label. This number is the current Program Counter inside the Method Frame Node.

Incoming edges

- $\langle PC \rangle$ edge from a Method Frame Node or the JVM Node

C.8 Stack Node

Description A Stack Node is used to model the stack of the JVM. Each Method Frame Node has it's own stack.

Representation A Stack Node is represented by a node without a self edge.

Outgoing edges

- $\langle previous \rangle$ edge to a Stack Node, to the previous item on the stack
- $\langle value \rangle$ edge to an Object Node, the value that is on the stack

Incoming edges

- $\langle previous \rangle$ edge from a Stack Node, from the next item on the stack
- $\langle current \rangle$ edge from an Object Node, indicating the current value on the stack

Appendix D

Implemented JVM instructions

This appendix gives an overview of the implemented JVM instructions. The grouping used here is the same as in Section 3.2.2. Instructions marked by a $\surd$ are implemented.

D.1 Stack operations

Instruction number	Instruction name	Implemented
01	aconst_null	
02	iconst_m1	
03	iconst_0	$\surd$
04	iconst_1	$\surd$
05	iconst_2	$\surd$
06	iconst_3	$\surd$
07	iconst_4	$\surd$
08	iconst_5	$\surd$
09	lconst_0	
10	lconst_1	
11	fconst_0	
12	fconst_1	
13	fconst_2	
14	dconst_0	
15	dconst_1	
16	bipush	
17	sipush	
18	ldc	
19	ldc_w	
20	ldc2_w	
87	pop	
88	pop2	
089	dup	$\surd$
090	dup_x1	

091	dup_x2	
092	dup2	
093	dup2_x1	
094	dup2_x2	
095	swap	

D.2 Arithmetic operations

Instruction number	Instruction name	Implemented
096	iadd	
097	ladd	
098	fadd	
099	dadd	
100	isub	
101	lsub	
102	fsub	
103	dsub	
104	imul	
105	lmul	
106	fmul	
107	dmul	
108	idiv	
109	ldiv	
110	fdiv	
111	ddiv	
112	irem	
113	lrem	
114	frem	
115	drem	
116	ineg	
117	lneg	
118	fneg	
119	dneg	
120	ishl	
121	lshl	
122	ishr	
123	lshr	
124	iushr	
125	lushr	
126	iand	
127	land	
128	ior	
129	lor	

130	ixor	
131	lxor	
132	iinc	
148	lcmp	
149	fcmpl	
150	fcmpg	
151	dcmpl	
152	dcmpg	
190	arraylength	

D.3 Control flow operations

Instruction number	Instruction name	Implemented
153	ifeq	
154	ifne	
155	iflt	
156	ifge	
157	ifgt	
158	ifle	
159	if_icmpeq	
160	if_icmpne	
161	if_icmplt	
162	if_icmpge	
163	if_icmpgt	
164	if_icmple	
165	if_acmpeq	
166	if_acmpne	
167	goto	
168	jsr	
169	ret	
170	tableswitch	
171	lookupswitch	
172	ireturn	
173	lreturn	
174	freturn	
175	dreturn	
176	areturn	
177	return	√
191	athrow	
194	monitorenter	
195	monitorexit	
198	ifnull	√
199	ifnonnull	√

200	goto_w	
201	jsr_w	

D.4 Load and store operations

Instruction number	Instruction name	Implemented
21	iload	
22	lload	
23	fload	
24	dload	
25	aload	
26	iload_0	√
27	iload_1	√
28	iload_2	√
29	iload_3	√
30	lload_0	
31	lload_1	
32	lload_2	
33	lload_3	
34	fload_0	
35	fload_1	
36	fload_2	
37	fload_3	
38	dload_0	
39	dload_1	
40	dload_2	
41	dload_3	
42	aload_0	√
43	aload_1	√
44	aload_2	√
45	aload_3	√
46	iaload	
47	laload	
48	faload	
49	daload	
50	aaload	
51	baload	
52	caload	
53	saload	
54	istore	
55	lstore	
56	fstore	
57	dstore	

58	astore	
59	istore_0	
60	istore_1	
61	istore_2	
62	istore_3	
63	lstore_0	
64	lstore_1	
65	lstore_2	
66	lstore_3	
67	fstore_0	
68	fstore_1	
69	fstore_2	
70	fstore_3	
71	dstore_0	
72	dstore_1	
73	dstore_2	
74	dstore_3	
75	astore_0	
76	astore_1	√
77	astore_2	√
78	astore_3	√
79	iastore	
80	lastore	
81	fastore	
82	dastore	
83	aastore	
84	bastore	
85	castore	
86	sastore	

D.5 Field access operations

Instruction number	Instruction name	Implemented
178	getstatic	√
179	putstatic	√
180	getfield	
181	putfield	√

D.6 Method invocation operations

Instruction number	Instruction name	Implemented
182	invokevirtual	√

183	invokespecial	√
184	invokestatic	√
185	invokeinterface	

D.7 Object allocation operations

Instruction number	Instruction name	Implemented
187	new	√
188	newarray	
189	anewarray	
197	multianewarray	

D.8 Conversion and type checking operations

Instruction number	Instruction name	Implemented
133	i2l	
134	i2f	
135	i2d	
136	l2i	
137	l2f	
138	l2d	
139	f2i	
140	f2l	
141	f2d	
142	d2i	
143	d2l	
144	d2f	
145	i2b	
146	i2c	
147	i2s	
192	checkcast	
193	instanceof	
196	wide	

D.9 Other operations

Instruction number	Instruction name	Implemented
00	nop	
186	xxxunusedxxx1	

D.10 Reserved operations

Instruction number	Instruction name	Implemented
202	breakpoint	
254	impdep1	
255	impdep2	

Appendix E

CD-ROM

The CD-ROM includes the source of the implemented translator. See the README file on the CD-ROM for more information.