Software Metrics
as
Class Graph Properties

Master of Science thesis

H. Kastenberg

Enschede
July 2004

Graduation committee: prof. dr. ir. M. Akşit
dr. ir. A. Rensink
dr. ir. K. G. van den Berg

Chair: Software Engineering
Department: Electrical Engineering, Mathematics and Computer Science
University: University of Twente
Summary

Software metrics are useful for multiple purposes, such as increasing software quality and identifying error-prone software entities. Many modelling techniques, however, do not provide formal means for expressing metrics as properties of models.

This report focuses on the definition and construction of class graphs for which we will express particular properties in a formalism called Description Logics.

We have specified graph grammars for transforming the abstract syntax tree of a program into the corresponding class graph. After defining the Description Logic language we have expressed some selected metrics in this language. The expression evaluator we implemented checks whether the class graph contains elements that satisfy the expressions. The evaluator has been implemented in an ad-hoc way and needs to be systematically redesigned in the future.

Description Logics are a useful formalism for expression properties of graphs as long as those properties do not require mathematical computations.
# Table of Contents

1 Introduction .................................................. 1
   1.1 Program Characteristics .................................. 1
   1.2 Problem Description and Approach ......................... 3
   1.3 Overview ................................................. 4

2 Modelling ....................................................... 5
   2.1 Definitions ................................................ 5
   2.2 Running example .......................................... 7
   2.3 Currently used models ..................................... 9
   2.4 Class Graphs .............................................. 19
   2.5 Summary .................................................. 27

3 Software Metric Definitions .................................. 29
   3.1 Software Metrics ........................................ 29
   3.2 Description Logics ....................................... 33
   3.3 Metrics in Description Logics ............................ 41
   3.4 Summary .................................................. 46

4 Model Construction .............................................. 47
# Table of Contents

4.1 Model Construction Mechanisms ........................................ 47
4.2 AST-visitors ..................................................................... 53
4.3 Graph Grammars ............................................................. 53
4.4 Summary ........................................................................ 63

5 Software Metric Calculation .................................................. 65
5.1 Expression Evaluation ....................................................... 65
5.2 Running the tool ............................................................... 67
5.3 Summary ........................................................................ 73

6 Discussion and Conclusion ................................................... 75
6.1 Evaluation ........................................................................ 75
6.2 Related Work .................................................................... 76
6.3 Future Work ..................................................................... 77

Bibliography ......................................................................... 81

A ANTLR ............................................................................... 83
A.1 Installing ANTLR .............................................................. 83
A.2 Running ANTLR ............................................................... 83
A.3 ANTLR Java 1.4 Grammar ................................................ 84

B GROOVE ........................................................................... 97
B.1 Installing GROOVE .......................................................... 97
B.2 Using GROOVE .............................................................. 97
B.3 Syntax ........................................................................... 98

C Description Logic Grammar ................................................ 101
C.1 description-logic.g .......................................................... 101

D Description Logic Syntax .................................................... 105
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>E Graph Production Rules</strong></td>
<td></td>
</tr>
<tr>
<td>E.1 Control Flow GPS</td>
<td>107</td>
</tr>
<tr>
<td>E.2 Call GPS</td>
<td>110</td>
</tr>
<tr>
<td>E.3 Dependence GPS</td>
<td>112</td>
</tr>
<tr>
<td><strong>F Abstract Syntax Trees of DL-expressions</strong></td>
<td></td>
</tr>
<tr>
<td>F.1 Fan-in and Fan-out</td>
<td>113</td>
</tr>
<tr>
<td>F.2 Cyclomatic Complexity</td>
<td>114</td>
</tr>
<tr>
<td>F.3 Lack of Cohesion</td>
<td>115</td>
</tr>
<tr>
<td><strong>G Results Prometrix</strong></td>
<td></td>
</tr>
<tr>
<td>G.1 calculator.pas</td>
<td>117</td>
</tr>
<tr>
<td>G.2 Call graph information</td>
<td>119</td>
</tr>
<tr>
<td>G.3 Flow graph information</td>
<td>120</td>
</tr>
<tr>
<td>G.4 Summary</td>
<td>121</td>
</tr>
<tr>
<td><strong>H CD-ROM</strong></td>
<td>123</td>
</tr>
</tbody>
</table>
TABLE OF CONTENTS
Introduction

Software metrics are used to get insight into different kinds of software quality factors, being high-level external attributes like reliability, usability and maintainability, and their relation with internal attributes, such as size, coupling and complexity \[10\]. Some of the concrete proposals in literature for measuring software quality take the form of a so-called metric suite (i.e. a set of measures that aim to give a balanced judgement of quality). Existing tools for software development such as Rational Rose and Borland Together contain functions to compute numbers indicating special characteristics of a given design or implementation, concerning for example complexity, cohesion or coupling.

Computing those values requires specific models of (part of) software artifacts such as specification, design or source code. Many models from which software metric values are determined can be viewed upon as being graphs: models containing certain objects and relations between them (in graph terminology called nodes and edges, respectively). In order to be able to compute particular metric values, the model must conform to some metamodel, which describes its structure.

In this project we concentrate on how to express metrics as properties of graphs. This will be done by using a general logic for reasoning about knowledge, called Description Logics.

1.1 Program Characteristics

The software development process is based on the creation of software artifacts. The UML \[20\] is a widely applied modelling language for creating software artifacts. UML models, such as class diagrams and interaction diagrams, have a prescriptive character in that they prescribe the structure and behaviour, respectively, of the system under development. One
particular software artifact is the program source. From the program source a number of program characteristics can be determined, like the total number of lines of code or the code-comment ratio.

McCall [16] describes a software quality model which consists of three layers: factors, criteria and metrics. Quality factors, such as reliability and usability, are expressed in terms of lower-level criteria, such as structuredness and traceability. Metrics serve as actual measures for the criteria and therefore indirectly measure high-level quality factors.

In this project we focus on metrics. Software metrics have been defined to assign values to various program characteristics. Getting from the software artifact to values which give insight in particular characteristics requires a couple of steps to be taken. Those steps are shown in Figure 1.1. If the characteristics to be analysed are known, it is necessary to define what model(s) will be constructed in order to measure them. These models define abstractions of the software artifact with respect to the specific characteristics. The relation between the artifact and the model must be specified, for example by specifying a mapping from an artifact to a model. Finally, a mapping needs to be specified from the model to the set of values on which the metrics are defined. The metric values are not useful if there is no way of ordering them. Therefore, an ordering function must be defined on the set of metric values. Usually this set consists of the real or natural numbers.[17]

Figure 1.1 forms the starting point of the project. It is assumed that the program source is available. Taking the source, some suitable models will be constructed. Having those models, the goal is to be able to express certain properties using some general logic. These logic expressions can be used to determine particular values as being properties of that model.

As the figure shows, there are two different kinds of transformations, both represented as an arrow. The first one is the construction of models out of the program source. There are many different aspects of source code that can be modelled. In this project the focus is on modelling static structures. The second arrow denotes value-determination given a particular model using specific metric definitions.
1.2 Problem Description and Approach

Since many software modelling techniques do not provide formal means for expressing properties of the models, this project aims at combining a modelling technique with a formalism for property expression.

We express different properties of a particular software artifact. The software artifact of subject is the program source in an object-oriented programming language. We develop a particular model, which we call a class graph. As the name ‘class graph’ indicates, this model is a graph representing specific characteristics of a single class. A class graph has a descriptive character, in contrast to UML models, in that it describes particular aspects of the program source. The decision to use a graph-representation is based on the fact that graphs are a good starting point for formal processing.

The characteristics being modelled by a class graph represent structural information from a static point of view upon the system: structural information which is known at compile-time. Modelling dynamic behaviour of programs is not in the scope of the project, defined in this thesis. A class graph will contain information about control flow of methods, calling relations between methods and dependencies between methods.

We will enumerate a number of concepts concerning graphs in general and some specifically related to class graphs. The construction of class graphs is done by performing graph transformations, taking the abstract syntax tree of a program as a starting point. By transforming the abstract syntax tree we add structural information to it concerning control flow, call and dependence information.

The properties being expressed are related to the static structures being modelled in a class graph. We select four metrics, three of which are defined at method-level. Concerning the control flow structure of methods we express their cyclomatic complexity. Next to that, we express the fan-in and fan-out of methods which give insight in the calling relations between methods. At class-level we will express the lack of cohesion in methods as a property of a class graph, which can be derived from the dependence relations between methods.

The formalism that will be used for property expression is called Description Logics. Description Logics are based on first order logic, but provide a more intuitive syntax for expressing properties than first order logics do. After expressing those different properties in a Description Logic language, we evaluate the expressions. The result of the evaluation will afterwards be compared with values calculated by other software development tools. How these values must be interpreted in relation with high-level quality factors, such as maintainability and reliability, is not in the scope of this project.
1.3 Overview

The program source will be taken as a starting point for the assignment. Therefore the left-most element of Figure 1.1 not be discussed in a distinct chapter. The other four elements of it are each related to a single chapter.

Chapter 2 discusses the different concepts. This chapter also specifies the abstraction from the program to its class graph. Chapter 3 defines the selected metrics and the formalism in which they will be expressed. The transformation from a program source to its class graph is subject of Chapter 4. Chapter 5 discusses the evaluation of the metric expressions and compares the result with output of other software development tools for equivalent metrics. At the end of this report we will evaluate the results of the project and discuss a number of major decisions that have been made.
This chapter discusses the abstraction from the program(-source) to specific models. It gives an overview of the different modelling techniques that will be used in the remainder of this report.

First we will need a number of definitions concerning graphs in general. We mention type graphs as a way to specify the structure of graphs. Then, four different models will be distinguished, namely abstract syntax trees, control flow graphs, call graphs and dependence graphs. For each model we explain what is being modelled by it and what goal it serves. Thereafter, we introduce a new concept called class graph and motivate why we need it. A class graph puts all four models just mentioned together into one single model. For each of the four models we will discuss the way they are related to each other in the class graph.

In order to make things more concrete, we use an example program on which the different concepts will be applied.

2.1 Definitions

The models used in this assignment are all viewed upon as being graphs. Graphs and subgraphs will play a very important role in this assignment. Formal definitions and a number of notational conventions are given below.
Definition 2.1 (graph) A graph $G$ is a pair $(N, E)$ where
- $N \neq \emptyset$ is the set of nodes (also called vertices)
- $E \subseteq N \times L \times N$ is the set of edges where $L$ is the set of labels

Definition 2.2 (subgraph) Given a graph $G = (N, E)$, another graph $G' = (N', E')$ is a subgraph of $G$ iff
- $N' \subseteq N$ and
- $E' \subseteq E$

About graphs in general, a couple of related concepts and their notation are enumerated below [22].

- The set of all graphs will be denoted $\mathcal{G}$, ranged over by $G, H$.
- The set of all nodes will be denoted $\mathcal{N}$, ranged over by $p, q$.
- The set of all labels will be denoted $\mathcal{L}$, ranged over by $a, b$.
- Given a graph $G$, the set of edges of $G$ is denoted $E_G$, the set of labels of $G$ is denoted $L_G$ and the set of nodes of $G$ is denoted $N_G$
- Given an edge $e = (p, a, q) \in E$, $p, q$ and $a$ are called the source, target and label of $e$, denoted $\text{src}(e)$, $\text{tgt}(e)$ and $\text{l}(e)$
- For a given edge $e$, the source and target of it are either different or equal; in the latter case, the edge is called a self-edge of the concerning node
- For a given edge $e$, its label can either be a single the source and target of it are either different or equal; in the latter case, the edge is called a self-edge of the concerning node
- A path is a sequence of consecutive (directed) edges, some of which may be traversed more than once [10].
- Given a graph $G$, two nodes $p, q \in N_G$ are directly connected if there exists an edge $e \in E_G$ such that $p = \text{src}(e)$ and $q = \text{tgt}(e)$ or vice versa; two nodes $p$ and $q$ are indirectly connected if there exists a path from $p$ to $q$ (not considering the direction of the edges in between).

\footnote{By representing an edge as being a triple of two nodes and a label, a graph can have multiple, distinctly labelled edges between two nodes.}
Definition 2.3 (morphism) Let G and H be two graphs. A morphism from G to H is a pair of partial functions \( f = (f_N, f_E) \) with \( f_N : N_G \rightarrow N_H \) and \( f_E : E_G \rightarrow E_H \), such that the following partial confluence properties hold:

- \( tgt_H \circ f_E \subseteq f_N \circ src_G \);
- \( tgt_H \circ f_E \subseteq f_N \circ tgt_G \);
- \( l_H \circ f_E \subseteq l_G \).

The morphism is called injective [total] if both \( f_N \) and \( f_E \) are injective [total].

We will write \( f : G \rightarrow H \) to indicate that \( f \) is a morphism from \( G \) to \( H \).

Since we use graphs for modelling specific program characteristics, we need a way to specify their structure. This way of meta-modelling for graphs is called graph typing [23].

Definition 2.4 (typing, type graph) Let \( G \in \mathcal{G} \) be arbitrary. A typing of \( G \) is a morphism \( \tau : G \rightarrow T \), where \( T \in \mathcal{G} \) is called a type graph.

When using these concepts, \( T \) is called a type of \( G \) and \( G \) an instance of \( T \).

Type graphs do not contain any information about restrictions of occurrences of elements and the number of edges allowed between different types of nodes (so called cardinality). If there are special restrictions, this must be mentioned explicitly.

2.2 Running example

In this report a number of existing concepts will be used and some new ones will be introduced. When appropriate, a simple example will be used to make things more concrete. The source given in Listing 2.1 implements a simple calculator written in Java, (parts of) which will be used for clarification of the different existing and new concepts.

```java
import java.io.*;

class Calculator {
    int a, b;
    static int result;

    public Calculator(int x, int y) {
        setA(x);
        setB(y);
    }
}
```

Harmen Kastenberg

7
```java
public static void main(String args[]) throws IOException {
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    String oper = null;

    System.out.println("Enter the operation to be performed: ");
    oper = in.readLine();

    while (!oper.equals("stop")) {
        System.out.println("Enter the first integer-value: ");
        int x = Integer.parseInt(in.readLine());
        System.out.println("Enter the second integer-value: ");
        int y = Integer.parseInt(in.readLine());

        Calculator calc = new Calculator(x, y);

        if (oper.equals("add"))
            calc.add();
        else if (oper.equals("sub"))
            calc.sub();
        else if (oper.equals("mul"))
            calc.mul();
        else
            System.out.println("Operation unknown, try again...");

        System.out.println("Result of operation: "+ result);
        System.out.println("Enter the operation to be performed: ");
        oper = in.readLine();
    }
}

public void add() {
    result = a + b;
}

public void sub() {
    if (a > b)
        result = a - b;
    else
        result = b - a;
}

public void mul() {
    result = a * b;
}

public void setA(int i) {
    a = i;
}

public void setB(int i) {
    b = i;
}
```

Listing 2.1: Calculator.java
2.3 Currently used models

There are many different ways to model static structures of a program. A selection of them are enumerated in Table 2.1 and will be discussed here, since they will play an important role in this project.

<table>
<thead>
<tr>
<th>Model</th>
<th>Properties being modelled</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract Syntax Tree</td>
<td>Program structure</td>
</tr>
<tr>
<td>Control Flow Graph</td>
<td>Procedural control flow</td>
</tr>
<tr>
<td>Call Graph</td>
<td>Calling relation between procedures</td>
</tr>
<tr>
<td>Dependence Graph</td>
<td>Dependency relations between statements</td>
</tr>
</tbody>
</table>

Table 2.1: The various models playing an important role in this report.

Each of those models can be represented as a graph. For each model we will explain how to look upon it when representing it as a graph: what the nodes and edges are related to and what it means for two nodes being connected by an (directed) edge. For every structure a type graph will be given and briefly explained.

2.3.1 Abstract Syntax Trees

When the source of a program is available together with the grammar it conforms to, the corresponding abstract syntax tree can be constructed. Abstract syntax trees are used to visualize program source in an intuitive way.

Trees are a special kind of graphs, namely directed acyclic graphs, often abbreviated to ‘dag’. Every node in a tree has at most one incoming edge and there exists exactly one node without incoming edges, called its root.

The name ‘abstract syntax tree’ implies that it can also be modelled as a directed acyclic graph. When viewing upon abstract syntax trees as being graphs, there is a relation between the elements of the abstract syntax tree and the source-code of the program.

Nodes of an abstract syntax tree, on the one hand, can be split up in two different categories: source-nodes and grammar-nodes. Source-nodes can directly be mapped to the source-code of the program, while grammar-nodes determine the structure of the tree without having a direct relation to the source-code; they form the intermediate levels from the root of the abstract syntax tree and the source-nodes. Another categorization can be applied by splitting the nodes into terminal nodes (also called leaves) and non-terminal nodes. These
two categorizations define the same partitioning: the set of source-nodes equals the set of terminal-nodes and the set of grammar-nodes equals the set of non-terminal nodes.

Edges of an abstract syntax tree, on the other hand, form the coupling between these different kinds of nodes. Edges do not have to be labelled, they only connect two nodes. In this project, however, all edges are labelled; Definition 2.1 says that edges are triples of two nodes and one label. Since the edges of a abstract syntax tree are directed, one can distinguish between the source- and the target-node.

Since the structure of the abstract syntax tree is determined by the corresponding grammar, it can be described by a type graph. Specifying that type graph, however, is very time consuming because of the size of the grammar. Since the grammar describes the structures of the language, it can also be seen as the specification of the type graph of the abstract syntax tree. Take for example the following grammar rule listed in Listing 2.2. (The syntax of ANTLR will not be discussed in detail in this paragraph. We only address the parts that are important concerning the construction of the abstract syntax tree.)

```java
// Definition of a Java class
classDefinition : [AST modifiers] :
    "class" IDENT
    // it might have type parameters
    (typeParameters)?
    // it might have a superclass...
    sc: superClassClause
    // it might implement some interfaces...
    ic: implementsClause
    // now parse the body of the class
    cb: classBlock
{ #classDefinition = #(#[CLASS_DEF,"CLASS_DEF"],modifiers,IDENT,sc,ic,cb); }
```

Listing 2.2: Grammar rule which describes the structure of a Java class

The important element of this grammar rule is the tree construction command in Line 12. It says that the subtree constructed by this rule consists of a root-node labelled 'CLASS_DEF', having the other elements as its children. The part of the type graph belonging to this grammar rule is shown in Figure 2.1.

The rough framework of the abstract syntax tree relates to the non-terminals in the grammar specification: the grammar-nodes. A couple of possible labels of grammar-nodes are

1. When we talk about the source-node of an edge we mean something different than when we talk about a node being of type source-node. The first concept refers to the occurrence of the node with respect to outgoing edges from that node, the second concepts defines that there is a direct relation between this node and the source-code of the program.

2. Nodes themselves are not labelled. When we talk about the label of a node, we actually mean the label of the edge pointing from that node to itself, its so called self-edge.
2.3 Currently used models

Figure 2.1: A small part of the type graph of the abstract syntax tree.

enumerated in Table 2.2

<table>
<thead>
<tr>
<th>Label</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLASS_DEF</td>
<td>'EXPR'</td>
</tr>
<tr>
<td>METHOD_DEF</td>
<td>METHOD_CALL</td>
</tr>
<tr>
<td>VARIABLE_DEF</td>
<td>'WHILE'</td>
</tr>
<tr>
<td>PARAMETERS</td>
<td>'IF'</td>
</tr>
<tr>
<td>BLOCK</td>
<td>'FOR'</td>
</tr>
</tbody>
</table>

Table 2.2: Example labels of self-edges of grammar-nodes in abstract syntax trees.

Figure 2.2 shows the abstract syntax (sub)tree of the method sub().

2.3.2 Control Flow Graphs

The control flow of methods can be modelled using control flow graphs. The control flow of a particular method is visualized in the form of a graph.

One of the applications of control flow graphs is the determination of the cyclomatic complexity of procedures. McCabe [15] defined the cyclomatic number $v$ (or cyclomatic complexity) of a procedure by taking its flowgraph $F$ as

$$v(F) = e - n + 2,$$

where $F$ has $e$ edges and $n$ nodes.

Before discussing control flow graphs in more detail, the formal definition is given.

**Definition 2.5 (flow graph)** [10] A flow graph is a directed graph in which two nodes, the start node and the stop node, have special properties: the stop node has out-degree...
Figure 2.2: Syntax tree of the method `sub()` of class Calculator.
Currently used models

zero, and every node lies on some path from the start node to the stop node.

The nodes of a control flow graph (excluding the stop node) represent executable statements; the edges represent the sequential relation between statements. Each node in a control flow graph on the path from the start node to the end node (including the start node, excluding the end node) is either a predicate- or a procedure-node. The difference between them is that a predicate-node refers to the evaluation of a conditional expression, while a procedure-node refers to all other kinds of expressions (assignments, method-calls, etc.). Since conditional expressions evaluate to either true or false, a predicate node has two outgoing edges, in contrast to procedure-nodes, which have one outgoing edge. From the definition it is clear that every control flow graph has one start and one end node. The end node is the only node of a control flow graph not having any outgoing edges. In this project we abstract from control flow concerning exception handling.

One way of formalising the structure of control flow graphs is by defining its type graph. This is done in Figure 2.3.

This type graph states that a control flow graph consists of several nodes being related to each other by edges labelled ‘next’. Predicate- and procedure-nodes are labelled ‘pred’ and ‘proc’, respectively, and may have an edge to other nodes of the same or the other type. The number of outgoing edges of particular nodes has already been discussed in the previous paragraph, but this can not be derived from the type graph. The type graph also explicitly shows that the end node of a control flow graph does not have outgoing edges labelled ‘next’.

The notion of control flow graphs can best be explained by giving a short example.

---

1 Another name for the stop node.
2 In this report we focus on boolean conditional expressions. A consequence of this decision is that we cannot take case-expressions into account. For case-expressions, the predicate-node representing the branch-condition can have outdegree greater than two.
Example 2.1 Taking the method `main(String[])` from Listing 2.1, Figure 2.4 shows the corresponding control flow graph. The numbers in the figure refer to line numbers in the listing.

The object oriented programming paradigm distinguishes between methods and constructors. Constructors are used for creating objects while the execution of procedures generally only change the state of a particular object. Since constructors contain statements for initializing objects, there exists a control flow graph representing their control flow. Therefore, from this point on, we will not distinguish between the control flow graph of a method or constructor. When we refer to a procedure this can either be a method or a constructor.

2.3.3 Call Graphs

A call graph models calling relations between modules in a program. From a call graph it becomes clear which procedures call others.

By examining call graphs it is possible to derive whether particular modules provide specific or generic functionality: if a module is called by very few other modules, this might indicate that its functionality is too specific. On the other hand, if a module is called by many other modules, it may be too generic.

The definition of call graphs is taken from [10].

\[^{1}\] IOException control flow is not captured.
2.3 Currently used models

Definition 2.6 (call graph) A call graph is a directed graph representing the calling relation between a program’s modules. It has a distinguished root node, corresponding to the highest-level module and representing an abstraction of the whole system.

In this project a module is either a method or a constructor (i.e. a procedure as defined in the previous section about control flow graphs)

The nodes of a call graph represent the procedures being either callees or callers; their edges represent the calling relations between the procedures. Since call graphs are directed graphs, every edge has an explicit source and target node, representing the calling and called procedure, respectively. Cycles in a call graph represent recursion.

One issue related to call graphs is the fact that static and instance variables can be initialised by use of a method-call. Such method-calls are not contained in a procedure. According to The Java Language Specification [29] the Java Virtual Machine executes these statements as part of the constructor. In this project, however, method-calls of this category are not in the scope of the call graphs.

Analogous to control flow graphs, the structure of call graphs can be defined by a type graph. Figure 2.5 shows the type graph of call graphs when looking upon them from the most general perspective.

![Figure 2.5: Type of call graph.](image)

This type graph states that a call graph consists of nodes representing procedures that can call each other. In this project, however, the focus is on call graphs representing the calling relation between procedures being defined in a single class. These procedures, however, can still call procedures in libraries. Therefore, the type graph that will serve as a reference point in this report, shown in Figure 2.6, is a little different from the type graph shown in Figure 2.5 in that it explicitly represents calls to procedures in library-classes.

From this type graph it is not clear how many edges may exist between two procedures if one procedure calls another one (or itself). It is not interesting whether a procedure contains
Figure 2.6: Type of call graph explicitly representing calls to external method.

one or even thousand statements calling the other procedure. In both cases we only need one edge representing that there exists at least one statement in which the other procedure is called.

Example 2.2 The call graph of the class Calculator is shown in Figure 2.7. It contains elements with solid and with dashed lines. The solid elements represent the calls to methods of the class itself, the dashed elements refer to methods of library-classes. From this call graph it becomes clear that the method main(String[]) calls the methods add(), sub() and mul() being defined in this class, the methods equals(...), readLine() and println(...) and the constructors BufferedReader(...) and InputStreamReader(...) from library-classes and the constructor Calculator(), which calls the methods setA(int) and setB(int). □

Figure 2.7: Call graph of method main(String[]) in class Calculator.

2.3.4 Dependence Graphs

When talking about dependence graphs, a distinction is made between data dependence graphs and control dependence graphs. Data dependence graphs arise from the fact that several statements access or update the same (instance) variables and therefore the result
Currently used models

of the execution of one statement may depend on the result of other statements that have
been executed just before. Control dependence graphs originate from conditional relations
between statements. If a particular conditional expressions evaluates to \textit{true} it results in the
execution of statements that normally differ from the statements that would be executed
when the condition had evaluated to \textit{false}.

Program dependence graphs (PDG’s) are used for different purposes. \cite{11} describes how
PDG’s are used for program optimization. PDG’s are also used as the intermediate repre-
sentation for a vectorizing compiler as outlined in \cite{7}.

In this project we refer to dependence graphs as defined in \cite{7}.

\textbf{Definition 2.7 (dependence graph)} A \textit{dependence graph} is a directed graph \(G\), where
the vertices \(V\) (also called nodes) represent statements and predicate expressions and the
edges \(E\) represent both data values and control conditions on which operations at a vertex
depend.

The edges of dependence graphs may be partitioned into two subgraphs. Edges which rep-
resent data values on which a vertex depends form the \textit{data dependence subgraph}; edges
which represent control conditions on which a vertex depends form the \textit{control dependence
subgraph}.

From this perspective, it is not directly clear what the difference is between control flow
graph and control dependence graphs. The difference between them relies on the fact that
in a control flow graph the different paths that can be taken when executing a procedure are
modelled. Normally, a procedure has fixed points at which its execution stops. This point
is represented by the end node of the control flow graph. The control dependence graph, on
the contrary, does not model paths that can be taken when executing a procedure. It only
depicts whether the execution of statements (or blocks of statements) depend on a particular
conditional statement. This can best be explained using a very simple example taken from
\cite{2}.

\textbf{Example 2.3} When taking the method as shown in Listing 2.3, the control flow graph looks
like shown in Figure 2.8 (i). The corresponding control dependence graph is shown in Figure
2.8 (ii). The numbers of the nodes in the graphs refer to the line-numbers in the listing.
From the control flow graph it is possible to determine the number of control paths that can
be taken when executing a method (in this case 2). The control dependence graph only shows
whether particular executable statements depend on other statements. In this case the control
dependence graph shows that the statements on line 3 and 5 depend on the result of that on
line 2.  

\hfill \Box
In this project the focus is on program dependencies between statements at data-level. One important question about data dependencies is whether different procedures access common instance variables. This requires dependence information between statements occurring in those different procedures. In a regular data dependence graph, a statement directly refers to other statements it depends on. The edges of the data dependence graphs as they are used in this project do not directly point from one statement to another statement. The edges point from the variable being accessed in the statement to the place where that variable is declared. If one statement depends on another statement, they both have an edge to the same variable declaration.

In order to be able to reason about data dependencies in general at all, the structure of dependence graphs at data-level can not be arbitrary. The type graph in Figure 2.9 describes the structure of data dependence graphs.

This type graph describes that statements depending on others are connected to them by an edge. In this project, the data dependence graphs are modelled in a slightly different way. A dependency relation, for example, between two statements exists when both statements assign, or maybe only read, the same variable. The data dependence graph, as used in this
2.4 Class Graphs

In this project we aim at a modelling technique for modelling the discussed static program structures in one single model. We call such a model a *class graph*. The reason for introducing...
class graphs is because we want to be able to express different characteristics of the software artifact being modelled, the source code, using one consistent model. When using multiple models we need to keep those models pair-wise consistent, which can be very difficult.

The abstract syntax tree of a given program-source is taken as a starting point to transform it into the class graph. By performing graph-transformations, the original abstract syntax tree will be enriched with control flow, call and dependence information. In the rest of this section the concept of class graphs is discussed.

A class graph $G$ is a graph as introduced in Definition 2.1, for which we define the following additional concepts.

- There exists a partitioned set\(^1\) say $S_L$, out of $L_G$ such that every element of $S_L$ has its own characteristics. The elements of $S_L$ are called $L_{cflow}$, $L_{dep}$, $L_{call}$ and $L_{syntax}$.
- Since every edge $e \in E_G$ is related to exactly one label $l(e) \in L_G$, the partitioning of $L_G$ gives rise to a partitioned set $S_E$ of edges which is directly related to $S_L$. When $s_i(X)$ is the $i$th element of the partitioned set of the original set $X$, this relation can be defined as follows:

$$\forall e \in E_G : l(e) \in s_i(L_G) \rightarrow e \in s_i(E_G)$$

This relation implies that $E_G$ can also be partitioned into four subsets $E_{cflow}$, $E_{dep}$, $E_{call}$ and $E_{syntax}$.
- The partitioning of $L_G$ is based on the string representing the individual labels $l \in L_G$. Every label in the sets $L_{cflow}$, $L_{dep}$, $L_{call} \subseteq L_G$ is prefixed with the strings ”cflow:”, ”dep:” and ”call:”, respectively. The labels in the set $L_{syntax} \subseteq L_G$ do not have a common prefix.
- Each element $s_i(E_G)$ of the partitioned set $S_E$ itself forms another graph $H$. This graph $H$ is a subgraph of graph $G$ according to Definition 2.2. In the rest of this report these subgraphs will be denoted $G_{cflow}$, $G_{dep}$, $G_{call}$ and $G_{syntax}$.
- Every class graph contains exactly one node, say $n_{root} \in N_G$, which has no incoming edges

### 2.4.1 Relating subgraphs

In the previous paragraph, we distinguished four subgraphs in a class graph. As we will describe in Chapter 4, the $G_{syntax}$-subgraph defines the basic structure of a class graph. In

\(^1\)A partitioned set $S$ is a set of subsets of a given set such that all elements of $S$ are non-empty and disjoint and the union of those elements results in the original set (formally, $s_i \cap s_j = \emptyset$, for $i \neq j$ and $\bigcup_{i=1}^{n} s_i = S$, in which $s_i$ is the $i$th element of the partitioned set $S$ containing $n$ elements).
the following paragraphs we discuss how the other three subgraphs relate this basic structure. For each subgraph we will show a small part of the class graph which depicts the relation between the concerning subgraph and the basic structure. The construction of the class graph will be discussed in detail in Chapter 4.

The figures referred to in this section contain elements depicted in different ways. The elements with solid lines refer directly to the elements of the corresponding type graphs as defined in the previous section; the elements with dashed lines are parts of the abstract syntax tree to which the elements of the different type graphs relate.

Control Flow Graphs

Figure 2.11 describes the structure of control flow graphs as they appear in class graphs. A class graph contains a control flow subgraph for every procedure being defined. The control flow graphs of separate procedures are not connected to each other. Since Java-classes in general have more than one procedure, the control flow subgraph of a class graph is likely to contain multiple instances of the corresponding type graph. This means that the control flow subgraph of a class graph will usually be a graph built up of multiple unrelated graphs.

![Control Flow Graph Diagram](image)

Figure 2.11: Embedding the control flow graph in the class graph.

Every instance of a control flow graph is related to exactly one node representing a procedure. This node has two outgoing edges: one edge labelled ‘cflow-begin’ and one edge labelled ‘cflow-end’. These edges connect the procedure definition node with the start and end node, respectively, of the corresponding control flow graph. The start node of the control flow graph is either labelled ‘cflow:PROC’ or ‘cflow:PRED’; the end node is always labelled ‘cflow:END’.

In the running example it is nice to see how the control flow graphs of the different procedures are related to one or several elements of the $G_{\text{syntax}}$-subgraph. Figure 2.12 shows how the control flow graph of the main-method is related to the basic structure. The left-most node labelled ‘METHOD_DEF’ is part of the basic structure.
Figure 2.12: Class graph of class Calculator in which the control flow graph elements are highlighted.
2.4 Class Graphs

Call Graphs

The relation of the call graph with the $G_{\text{syntax}}$-subgraph is described by Figure 2.13. If there exists an edge labelled ‘call:calls’ between two nodes representing two procedures (or from one node to itself in case of direct recursion), this edge represents a calling relation from the procedure represented by its source node to the procedure represented by its target node.

![Diagram of a call graph embedded in a class graph]

Figure 2.13: Embedding the call graph in the class graph.

Relating the call graph of the running example to the basic structure of the corresponding class graph is shown in Figure 2.14. In this case, the $G_{\text{call}}$-subgraph has exactly one root-node from which all other nodes in the class graph representing procedures are directly or indirectly reachable. In general, this is not the case. For example call subgraphs of class graphs of Java-classes that defines a data structure are very likely not to cover all procedures, since those procedures generally provide means for reading and writing the data structure’s attributes.
Figure 2.14: Class graph of class Calculator in which the call graph elements are highlighted.
Dependence Graphs

The graph in Figure 2.15 shows how the dependence graph is related to particular elements of the $G_{syntax}$-subgraph. This figure shows three different graphs. The graph in Figure 2.15 (i) depicts the usage of instance variables at a certain place in a procedure. The variable being accessed is declared at class-level. Figure Figure 2.15 (ii) shows how the access of local variables is represented. The variable is declared at local-level. The last figure shows that the variable being accessed is a parameter of the procedure.

(i) Usage of instance variable. 

(i) Usage of local variable. 

(i) Usage of parameter.

Figure 2.15: Embedding the dependence graph in the class graph.

Figure 2.16 gives an example of how variable access is modelled in a class graph. The variables $a$, $b$ and $result$ are instance variables. Both expressions access all three instance variables.
Figure 2.16: Class graph of class `Calculator` in which the dependence graph elements are highlighted.
2.5 Summary

We have given an overview of different modelling techniques for modelling static structures of the program source. For each of these models we have pointed out why they are important. We have shown that the structure of graphs can be specified by using type graphs. Thereafter we introduced the concept class graph and defined a number of related concepts. The most important concept was that for each class graph $G$ there exists a partitioned set $S_E$ for which the union of its elements equals the set $E_G$, the set of all edges of $G$. The elements of $S_E$ are the edges of a subgraph of $G$. At the end of this chapter we discussed how these four graphs are related to each other (i.e. how they form the class graph together).
Chapter 3

Software Metric Definitions

The goal of this project is to express software metrics as class graph properties. Questions like

- What kind of properties do you want to express?
- What kind of formalism will be used to express those properties?

will be answered in this chapter. The first section discusses a number of interesting properties. Those properties are software metrics related to the different static structures that have been mentioned in the previous chapter. We handle four metrics: the fan-in, fan-out, the cyclomatic complexity and the lack of cohesion in methods. These four metrics are based on the static structure of the program source, but each metric measures a different aspect and therefore these metrics together give a balanced overview of the static structures of a single class.

In the second section we discuss the formalism that will be used for expressing the selected metrics, namely Description Logics (generally abbreviated to DL). We will give a short overview what DL’s are generally used for. A simple example will be used to clearify the use of DL’s. After explaining how DL’s are applied in this project we show how we express the software metrics in this formalism. This chapter does not handle why this specific formalism is taken; this will be discussed at the end of this report.

3.1 Software Metrics

The software engineering process can be split up in several phases from requirements analysis to software delivery and maintenance. Software design is one of the phases in between.
Various modelling techniques are used for modelling software artifacts at different levels of abstraction. Those models serve different purposes. They may be used

1. as a starting point for the next phase in the software engineering process (e.g. from design to implementation);
2. as a reference point for getting an overview of the structure (or behaviour) of the system;
3. for measuring particular properties of the program source.

The last purpose mentioned above is what this assignment is about. Software metrics provide means for determining specific properties of software artifacts. Many different software metrics have been defined for all different phases in the software engineering process. The focus will be on metrics for the analysis of implementation of object-oriented software systems. These metrics measure different program properties. The most important are enumerated below and shortly explained [28].

- **Size** metrics measure the size of elements, typically by counting the elements contained within an artifact. For example, the number of methods in a class and the number of classes in a package.
- **Coupling** metrics measure the degree to which the elements in an artifact are connected to each other. Two objects $A$ and $B$ are coupled if a method of object $A$ calls a method or accesses a instance variable of object $B$.
- **Inheritance** metrics measure the degree of hierarchying classes. The depth of a class in the inheritance tree indicates its level of reusability.
- **Complexity** metrics measure the degree of connectivity between elements in an artifact. For instance, counting the number of method invocations among the methods within one class can be considered a measure of class complexity; counting the number of independent paths that can be taken when executing a method can be considered a measure of method complexity.
- **Cohesion** metrics measure the degree to which the elements (e.g. methods) are logically related

### 3.1.1 Definitions

There are lots of different software metrics discussed in literature. For code analysis, two metric suites often serve as a starting point, being the CK-Suite, introduced by Chidamber and Kemerer [8], and the MOOD-Suite, introduced by Abreu [11].
3.1 Software Metrics

In this project we focus on three specific software metrics:

- Method’s fan-in and fan-out
- Cyclomatic complexity
- Lack of Cohesion in Methods

Method’s fan-in and fan-out

Relations between methods are interesting characteristics of software. Knowing which method call others, gives insight in the functionality of methods. If methods are only used by one single other method, it may turn out that such methods provide too specific functionality and it might be better to include the functionality in the calling method.

The definitions of the fan-in and fan-out of methods given by Fenton [10] are not only based on the calling relations between them. For the fan-in and fan-out of a module he also takes data structure accesses and updates, respectively, into account.

In this project we use the following definition.

**Definition 3.1** The fan-in and fan-out of methods are defined as the number of local methods that call the method and the number of the local methods that are called by the method, respectively.

The value of those metrics play an important role in the maintenance phase of the software engineering process. When a method has a large fan-in, this indicates that this method is used by many other methods. This means that when changing such a method, it can be difficult to predict if the functionality provided to the methods calling this one is still correct. A large fan-in can also indicate that the method implements multiple functionalities, in which case it would be better to divide it into multiple methods, each with their own specific purpose. On the other hand, when the fan-out of a method is large, a maintainer of the system has to understand many other methods making maintenance harder and more time consuming.

In the rest of this report we will refer to this metric by writing ‘procedure’s fan-in’ and ‘procedure’s fan-out’ or just ‘fan-in’ and ‘fan-out’, because of the same reason as mentioned in the previous chapter.
Cyclomatic complexity

Another software metric is introduced by McCabe in [15], namely the cyclomatic complexity of a procedure. The cyclomatic complexity is one of the most popular software characteristics. It can be used for many different purposes. Some of them are enumerated below.

- **Code development risk analysis**: during the software-development phase, software complexity can be measured in order to get insight in the risks for testing and maintenance

- **Change risk analysis in maintenance**: while maintaining software systems, code complexity may increase. Measuring the complexity before and after performing maintenance helps to decide how to minimize new introduced risks

- **Test Planning**: since the cyclomatic complexity gives the number of independent paths of every method, this metric gives the minimum number of paths to be tested.

Calculating this metric value requires the construction of the control flow graph of the concerning procedure. If the control flow graph is available, there are three different ways to calculate the cyclomatic complexity:

**Definition 3.2** Given the flow graph $G$ of a procedure, then the cyclomatic complexity, $V(G)$, of that procedure is defined by either of the following three equations:

- $V(G) = P + 1$
  where $P$ is the number of predicate nodes in the flowgraph, with the restriction that the outdegree of these nodes must be exactly 2

- $V(G) = E - N + 2$
  where $E$ and $N$ are the number of edges and nodes of the control flow graph

- $V(G) = \text{the number of regions in } G$
  where regions are the areas bounded by edges in a planar graph (including the outer infinite large area) \[32\]

The value of this metric tells something about the number of different paths that can be taken when calling that procedure. When this value exceeds a particular threshold (the common threshold-value is 10), one may be tempted to revise the source-code and simplify the control structure of the corresponding procedure.
3.2 Description Logics

Lack of Cohesion in Methods

Cohesion is also an important concept in the object-oriented programming paradigm. If methods of a class do not access common instance variables (i.e. are unrelated), this may indicate that the corresponding class provides multiple unrelated functionalities. In such cases, the class may better be split up into multiple classes, each concentrating on a single functionality. In the case a class defines a data structure it is very likely for the methods being unrelated since they may provide means reading or writing the data structure’s attributes. Such classes, typically have single methods for reading and updating each attribute, which causes such methods to lack cohesion. In such cases you do not want to redesign the implementation.

The lack of cohesion in methods is a metric which determines the degree of methods being related to each other. It has been introduced in the CK-Suite, but the definition given by Chidamber and Kemerer is too ambiguous: it is not able to express maximum (all attributes are accessed by all methods) or minimum (all attributes are accessed by a single method) cohesion. Therefore the definition of Henderson-Sellers [14] is taken (see Definition 3.3).

**Definition 3.3** The lack of cohesion of methods (LCOM) measures the dissimilarity of methods in a class by attributes. Consider a set of \( m \) methods \( M_1, M_2, \ldots, M_m \) that access a subset of data attributes \( A_1, A_2, \ldots, A_a \). Let

\[
\mu(A_i) = \text{number of methods that access data attribute } A_i
\]

Then,

\[
\text{LCOM} = \frac{1}{a} \sum_{i=0}^{a} \mu(A_i) - m
\]

In this formula we do not distinguish between methods or constructors and refer to this metric as the ‘lack of cohesion in procedures’. The range of the formula are the values between 0 to 1, including the boundaries. This metric has value 1 in the case of maximum lack of cohesion and value 0 in the case of minimum lack of cohesion.

3.2 Description Logics

Description Logics are a common language for representing knowledge [4]. When representing knowledge using a formal language, there is always a distinction between *defining*
characteristics and assigning them to individuals in the application domain. The set of definitions of the characteristics of individuals is also called the terminology of the application domain. Assigning characteristics to individuals is done by assertions.

Characteristics of individuals can either be conceptual or relational: a conceptual characteristic defines a set of individuals all having a particular characteristic, a relational characteristic defines a relationship between individuals. When the conceptual and relational characteristics are defined they can be assigned to individuals in the application domain. Next to the definition of atomic concepts and relations, it is possible to define more complex ones. The different ways that are available for defining new concepts and relations depends on the DL-language being applied. The basic DL-language $\mathcal{AL}$ (attributive language), as being introduced by Schmidt-Schauß and Smolka [26], provides some elementary concept constructors. The DL-language that will be used in this report is extended with a number of constructors. Table 3.1 gives an overview of the constructors we use for defining new concepts.

Analogue to defining new concepts, there are also constructors for defining new relations using existing ones. Those are shown in Table 3.2. The symbols of these constructors are added to the DL-language-name in sub- or superscript. These tables also define the formal semantics of all constructors. The names of concepts start with a capital letter, those of relations start with a regular small letter. Relations can only be of arity two (i.e. binary relations).

Extending the basic $\mathcal{AL}$-language with conceptual and relational constructors is expressed by the symbols shown in the tables. The $\mathcal{ALUE}$-language, for example, extends the $\mathcal{AL}$-language with the union constructor and the existential quantification constructor at conceptual level.

In order to define a formal semantics of $\mathcal{AL}$-concepts and -relations, the interpretation $\mathcal{I}$ is considered that consists of a non-empty set $\Delta^\mathcal{I}$ (the domain of the interpretation) and an interpretation function which assigns to every atomic concept $C$ a set $C^\mathcal{I} \subseteq \Delta^\mathcal{I}$ and to every atomic relation $R$ a binary relation $R^\mathcal{I} \subseteq \Delta^\mathcal{I} \times \Delta^\mathcal{I}$.

A short example will clarify the use of concepts and relations and how to construct new concepts or relations using existing ones.
### Constructor Table 3.1: Constructors for defining new concepts using existing concepts and relations.

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Syntax</th>
<th>Semantic</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>Top</td>
<td>$\top$</td>
<td>$\Delta^I$</td>
<td>$\mathcal{AL}$</td>
</tr>
<tr>
<td>Bottom</td>
<td>$\bot$</td>
<td>$\emptyset$</td>
<td>$\mathcal{AL}$</td>
</tr>
<tr>
<td>Intersection</td>
<td>$C \sqcap D$</td>
<td>$C^I \cup D^I$</td>
<td>$\mathcal{AL}$</td>
</tr>
<tr>
<td>Union</td>
<td>$C \sqcap D$</td>
<td>$C^I \cap D^I$</td>
<td>$\mathcal{U}$</td>
</tr>
<tr>
<td>Negation</td>
<td>$\neg C$</td>
<td>$\Delta^I \setminus C^I$</td>
<td>$C$</td>
</tr>
<tr>
<td>Value restriction</td>
<td>$\forall R.C$</td>
<td>${a \mid \forall b. (a, b) \in R^I \rightarrow b \in C^I}$</td>
<td>$\mathcal{AL}$</td>
</tr>
<tr>
<td>Full existential quantifier</td>
<td>$\exists R.C$</td>
<td>${a \mid \exists b. (a, b) \in R^I \land b \in C^I}$</td>
<td>$\mathcal{E}$</td>
</tr>
<tr>
<td>Unqualified number restriction</td>
<td>$\geq nR$</td>
<td>${a \mid | {b \mid (a,b) \in R^I} \geq n}$</td>
<td>$\mathcal{N}$</td>
</tr>
<tr>
<td></td>
<td>$\leq nR$</td>
<td>${a \mid | {b \mid (a,b) \in R^I} \leq n}$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$= nR$</td>
<td>${a \mid | {b \mid (a,b) \in R^I} |= n}$</td>
<td></td>
</tr>
</tbody>
</table>

### Constructor Table 3.2: Constructors for defining new relations using existing concepts and relations.

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Syntax</th>
<th>Semantic</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>role name</td>
<td>$R$</td>
<td>$R^I \subseteq \Delta^I \times \Delta^I$</td>
<td>$R$</td>
</tr>
<tr>
<td>union</td>
<td>$R \sqcup S$</td>
<td>$R^I \cup S^I$</td>
<td>$\sqcup$</td>
</tr>
<tr>
<td>intersection</td>
<td>$R \sqcap S$</td>
<td>$R^I \cap S^I$</td>
<td>$\sqcap$</td>
</tr>
<tr>
<td>negation</td>
<td>$\neg R$</td>
<td>$\Delta^I \times \Delta^I \setminus R^I$</td>
<td>$\neg$</td>
</tr>
<tr>
<td>inverse</td>
<td>$R^{-}$</td>
<td>${(x,y) \mid (y,x) \in R^I}$</td>
<td>$^{-1}$</td>
</tr>
<tr>
<td>composition</td>
<td>$R \circ S$</td>
<td>${(x,y) \mid \exists z. (x,z) \in R^I \land (z,y) \in S^I}$</td>
<td>$\circ$</td>
</tr>
<tr>
<td>transitive closure</td>
<td>$R^+$</td>
<td>$\bigcup_{n \geq 1} (R^I)^n$</td>
<td>$+$</td>
</tr>
<tr>
<td>reflexive-</td>
<td>$R^*$</td>
<td>$\bigcup_{n \geq 0} (R^I)^n$</td>
<td>$*$</td>
</tr>
<tr>
<td>transitive closure</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Harmen Kastenberg 35
Example 3.1 When representing properties of people and relations between them, you can start with stating whether a particular person is a male or a female. This can be done by introducing three concepts: Person, Male and Female. In this example we introduce a couple of individuals, namely Peter, Ann, Jean, Philip, Mark and Maria. The fact that each of these individuals are persons, can be formalized through:

\[
\text{Person}(Peter) \quad \text{Person}(Ann) \quad \text{Person}(Jean) \\
\text{Person}(Philip) \quad \text{Person}(Mark) \quad \text{Person}(Maria)
\]

Another way of formalizing this is by using sets. A set related to a concept has dimension one, containing only elements directly referring to individuals; a set representing a relation contains tuples of individuals. In this case, the set Person would contain six elements:

\[
\text{Person} = \{Peter, Ann, Jean, Philip, Mark, Maria\}
\]

Next to this, it is explicitly stated that Ann and Maria are females:

\[
\text{Female} = \{Ann, Maria\}
\]

The third concept, Male, is defined by using the existing concepts Person and Female:

\[
\text{Male} = \text{Person} \sqcap \neg \text{Female}
\]

stating that all persons not being females are males:

\[
\text{Male} = \{Peter, Jean, Philip, Mark\}
\]

Assume that there exists a relationship parentOf which contains the following elements:

\[
\text{parentOf} = \{(Peter, Mark), (Ann, Philip)\}
\]

Using the existing concepts and relation just defined, it is possible to define new concepts Father and Mother:

\[
\text{Father} = \text{Male} \sqcap \exists \text{parentOf.Person} \\
\text{Mother} = \text{Female} \sqcap \exists \text{parentOf.Person}
\]

Now we can infer that Peter is an element of the concept Father and that Ann is one of the concept Mother.
3.2 Extending DL-languages

The DL $\mathcal{ALU\text{E}NC}_{R}^{-1}$ is the DL-language which extends $\mathcal{AL}$ with union ($\cup$), negation ($\neg$), full existential quantification ($\exists$) and unqualified number restriction ($\forall$) constructors at conceptual level and with inverse ($-1$), composition ($\circ$), transitive closure ($R^+$) and reflexive transitive closure ($R^*$) constructors at relational level.

The DL $\mathcal{ALU\text{E}NC}_{R}^{-1}$ is equivalent to the DL $\mathcal{ALNC}_{R}^{-1}$, since conceptual union ($\cup$) can also be expressed using conceptual intersection (included in $\mathcal{AL}$) in combination with negation ($\neg$) (DeMorgan’s law):

$$(C \cup D) \iff \neg C \cap \neg D$$

Concerning the existential quantifier ($\exists$), it is enough to have the negation constructor ($\neg$) in combination with the value restricter (already included in the $\mathcal{AL}$-language).

At relational level, it can be seen that the reflexive transitive closure includes the transitive closure: $R^+ = \bigcup_{n \geq 1} (R^T)^n$ and $R^* = \bigcup_{n \geq 0} (R^T)^n$ and therefore $R^* = (R^T)^0 \cup R^+$.

Putting this all together, we may state that the language used in this project can be specified as an $\mathcal{ALNC}_{R}^{-1}$-language.

3.2.2 Concrete Domains and Aggregate Features

The description logic just mentioned will be extended with a concrete domain and concrete and aggregate features. First the concept of concrete domains and concrete features will be explained. Thereafter, the aggregate features will be discussed.

Definition 3.4 (concrete domain) A concrete domain $\mathcal{D} = (\text{dom}(\mathcal{D}), \text{pred}(\mathcal{D}))$ consists of

- a set $\text{dom}(\mathcal{D})$ (the domain)
- a set of predicate symbols $\text{pred}(\mathcal{D})$

The concrete domain that will be used in this context consists of the set real numbers. The predicate symbols used in this assignment are $<$ (less than), $<=$ (less or equal than), $>$ (greater than), $>=$ (greater or equal than), $==$ (equal to).

---

1This section is based on the work of Franz Baader and Ulrike Sattler as they have worked out in [5].
The introduction of concrete domains has its impact on the syntax of DL-expressions. Let $N_C$, $N_R$ and $N_F$ be disjoint sets of concepts, relations and feature names. A feature $f \in N_F$ is partial function

$$f^I : \Delta^I \to \Delta^I \cup dom(D)$$

This definition states that every already existing relation will now also be a feature. Special features are functions mapping an element from the interpretation domain $\Delta^I$ to an element from the concrete domain, a real number.

In order to relate the DL-expressions to this concrete domain, a new way of constructing concepts will be introduced, namely by using the predicates. A concept being defined using a predicate $P$ looks as follows:

$$P(u_1, ..., u_n)$$

in $u_i = f_1...f_m$ is a feature chain. The semantics of a concept being defined in terms of a predicate and feature chains is defined as follows:

$$P(u_1, ..., u_n) = \{d \in \Delta^I \mid (u_1^I(d), ..., u_n^I(d)) \in P^D\}$$

In this project we use predicates of arity two (i.e. binary predicates).

The semantics of a feature chain $u = f_1...f_m$ is defined as follows:

$$u^I(a) = f_m(f_{m-1}(...f_1(a)...))$$

The functions defined in Table 3.3 can be used to construct a new feature $f' : \Delta^I \to dom(D)$ given a feature $f' : \Delta^I \to \Delta^I$.

| $\text{countSecond}(f)$ | $= \{(x, y) \mid y = | \{z \mid (x, z) \in f\} | \}$ |
| $\text{countFirst}(f)$ | $= \{(x, y) \mid y = | \{z \mid (z, x) \in f\} | \}$ |
| $\text{mapToConstant}(C, n)$ | $= \{(c, n) \mid c \in C \land n \in dom(D)\}$ |

Table 3.3: Functions for defining new features based on existing concepts or relations.

The function $\text{countSecond}$ and $\text{countFirst}$ create features with pairs $(x,y)$, where the relation $R$ contains $y$ pairs having $x$ as its first and second element, respectively. The $\text{mapToConstant}$-function creates a feature taking the concept $C$ and some element $n \in \text{dom}(D)$ mapping every element $c \in C$ on $n$.
When having two features mapping elements \( d \in \Delta^I \) on elements \( d' \in \text{dom}(D) \), the functions \( \text{add} \) and \( \text{sub} \) are defined as follows:

\[
\text{add}(f_1, f_2) = \{(x, n) \mid (x, n_1) \in f_1 \land (x, n_2) \in f_2 \land n = n_1 + n_2\}
\]

\[
\text{sub}(f_1, f_2) = \{(x, n) \mid (x, n_1) \in f_1 \land (x, n_2) \in f_2 \land n = n_1 - n_2\}
\]

Now the DL-language is extended with a set of aggregation functions \( \text{agg}(D) \). The introduction of aggregated features requires the definition of multisets.

**Definition 3.5** A multiset \( M \) over \( S \) is a mapping \( M : S \to \mathbb{N} \), where \( M(s) \), \( s \in S \), denotes the number of occurrences of \( s \) in \( M \).

An aggregated feature can be expressed using the following syntax:

\[
f_1...f_n\Gamma(R \circ f)
\]

in which \( \Gamma \in \text{agg}(D) \) is an aggregation function. The parameter of aggregation functions is the multiset being composed from a relation \( R \) and a feature \( f \). In this assignment, three aggregation functions are defined. They are listed below.

\[
\begin{align*}
\text{count}(M) &= \sum_{y \in M} M(y) \\
\text{sum}(M) &= \sum_{y \in M} M(y) \cdot y \\
\text{average}(M) &= \frac{1}{|M|} \text{sum}(M)
\end{align*}
\]

### 3.2.3 Representing graphs using Description Logics

Applying Description Logics as a formalism for representing graphs and expressing their properties requires a subtle shift in the way of looking upon graphs. Instead of representing a graph as a set of nodes together with a set of edges, the labels on the edges are taken as the names of the relations and the pairs of source- and target-nodes will be interpreted as the elements of the relations. Concepts are referred to by the labels of self-edges. In principle, every distinct label of a self-edge refers to a distinct concept. In this project, however, not all labels of self-edges are interesting. We focus on the labels of the grammar-nodes, representing for example the concepts \texttt{CLASS\_DEF}, \texttt{METHOD\_DEF} and \texttt{VARIABLE\_DEF}.
When two distinct nodes, say $n_1$ and $n_2$, are connected to each other, there is at least one relation containing the pair $(n_1, n_2)$. Since edges represent relations between two nodes, the arity of the relations can only be two.

Originally, the relations only say something about directly connected nodes. Most of the time it is needed to be able to check whether particular nodes are indirectly connected to each other by a specific path. Therefore, the constructors of Table 3.2 can be used to define new relations between nodes that are indirectly connected.

A feature in a graph is represented by an edge between two nodes for which the second node has a self-edge having a label that refers to an element in the concrete domain.

### 3.2.4 Description Logic Grammar

The way of expressing properties of graphs using the DL-language is defined using a grammar. The entire grammar-specification is shown in Appendix C. ANLTR (ANother Tool for Language Recognition) is used to construct a parser for parsing DL-expressions which conforms to this grammar. The grammar also contains actions for tree building. This is needed because the syntax tree of the DL-expression is used by the DL-expression evaluator as discussed in Chapter 5.

Grammars are generally defined in terms of terminals, non-terminals and a single start symbol. The start symbol in the DL-grammar is expression. Since extending the DL-language with a concrete domain and features and aggregates over that domain as described in did not result in enough expression power, it had to be extended with extra functions as discussed before. This lead to a number of problems.

The first problem was that the different functions each require a specific list of parameters. Next to this, the parameters may be of different types. For example, the function countFirst requires a relation as its only paramater, while the mapToConstant-function only requires a concept. The current grammar is not able to distinguish between parsing a relation or a concept which causes non-determinism. When hard-coding the names of the functions being used in this project together with the list of parameters they require, this problem is solved in a pragmatic way.

A second problem concerns the names of the functions. If we do not hard-code their names, the parser is not able to distinguish between simple features and the defined functions. This forces us to put the their names in the grammar specification. One major rule concerning this issue is shown in Listing 3.1.
### 3.3 Metrics in Description Logics

DL-expressions do not return *true* or *false*. DL-expressions can only be used to construct new concepts (sets of individuals) or relations (sets of pairs of individuals). The elements in the concepts and relations satisfy the conditions stated in the DL-expressions. This means that it is not possible to ‘ask’ whether a particular individual satisfies a set of conditions. It is only possible to construct the set of elements satisfying the given expression and afterwards check what individuals are elements of that set.

At this point, the selected of the metrics are defined and the way of using Description Logics for representing graphs and expressing their properties is explained. This section describes how the four selected metrics, as discussed in the previous sections, can be expressed as properties of class graphs using the extenden DL-language. We use the notation as shown in Table 3.1 and 3.2.

In order to keep the DL-expressions readable we first define a new concept called **Procs**, representing the set of nodes representing either methods or constructors.
3.3.1 Procedure’s fan-in and fan-out

Recall that the definitions of fan-in and fan-out we use in this project only refer to the calling relations between local procedures. The calling relations between local procedures and external procedures are not of interest.

If a method calls itself recursively, the node representing this procedure has two self-edges: one edge labelled ‘METHOD_DEF’ representing that this node is an individual in the concept METHOD_DEF and another edge labelled ‘call:calls’ representing the recursive calling relation of this method.

A procedure’s fan-in is defined as the number of procedures calling this procedure. This number is equal to the numbers of pairs in the relation calls having the node representing this procedure as its second element. The first element may be a node representing a distinct procedure or the same procedure.

For this metric it is only possible to construct a new set consisting of the individuals satisfying the stated conditions. In this case, the following expression constructs the set of procedures having a fan-in which is greater than two:

\[
> (\text{countFirst}(\text{call}:\text{calls}), \text{mapToConstant}(\text{Procs}, 2))
\]

A procedures fan-out is just the opposite: the number of procedures called by this procedure. Determining the fan-out of a procedure is done by counting the number of pairs in the relation calls having the node representing this procedure as its first element:

\[
> (\text{countSecond}(\text{call}:\text{calls}), \text{mapToConstant}(\text{Procs}, 2))
\]

3.3.2 Cyclomatic Complexity

The cyclomatic complexity of a single procedure can be calculated in three ways (see Definition 3.2). The formula used here is the one based on the number of conditional expressions in the procedure. In order to get this number, the number of predicate nodes in the control flow graph of the procedure must be determined. A suitable way to do this is by defining a relation PROC2PREDs which contains pairs \((n_i, n_j)\) in which \(n_i\) refers to an element of the concept PROC, which represents the procedure to which the control flow graph is related, and
n_j refers to a node in that control flow graph being a predicate node. Using this relation it is possible to determine the number of predicate nodes of the control flow graph of a particular procedure and thus its cyclomatic complexity.

The relation Procs2Preds is defined as follows:

\[ \text{Procs2Preds} = \text{cflow} : \text{begin} \circ (\text{cflow} : \text{next}^* \circ \text{cflow} : \text{PRED}) \]

As already mentioned, a common threshold for the cyclomatic complexity of complex systems is 10. Checking whether there exist procedures with a cyclomatic complexity exceeding this value is done by constructing the concept containing the individuals satisfying that condition. Since the running example does certainly not contain a procedure exceeding this value, a lower value will be taken here. The set of method definition nodes having a cyclomatic complexity greater than two is constructed using the following expression:

\[
> (\text{add}(\text{countSecond}(\text{Proc2Preds}), \text{mapToConstant}(\text{Procs}, 1)), \\
\text{mapToConstant}(\text{Procs}, 2))
\]

### 3.3.3 Lack of Cohesion in Procedures

The lack of cohesion in procedures is based on the fact that instance variables are accessed by a set of procedures. In order to express this dissimilarity between procedures, we need to define relations from which we are able to extract which procedures access a particular instance variable.

Keeping the DL-expressions readable requires a new relation to be introduced, representing the transitive closure of the abstract syntax tree structure:

\[ \text{totalTree} = (\text{first} \_ \text{child} \sqcup \text{next} \_ \text{siblings})^+ \]

From the formula in Definition 3.3 we can derive that we need to define three different features from which we should be able to calculate the lack of cohesion: one for each different term in the quotient of the formula. We will first construct the needed features for each term and afterwards discuss how these features must be composed in one DL-expressions in order to express the lack of cohesion. Since this metric is defined at class-level the resulting elements of the DL-expression will be elements of the concept CLASS_DEF.

The feature related to the term with constant value 1 can easily be created by passing the concept CLASS_DEF together with the constant value 1 as parameters to the function mapToConstant. We call this feature classdef2One.

\[
\text{classdef2One} = \text{mapToConstant} (\text{CLASS_DEF}, 1)
\]
The denominator of the formula is the most complex one. It is necessary, first to define a number of relations and features which on their turn enable the construction of the needed feature.

- a relation $\text{Procs2InstVars}$ which maps the nodes representing the methods on the nodes representing the instance variables. Since the nodes representing instance variables and methods are siblings in the abstract syntax tree and instance variables may be defined at an arbitrary place in the global scope, we have to ensure that we also map the nodes representing methods at the nodes representing instance variables that are declared after the methods:

$$\text{Procs2InstVars} = \left( (\text{METHOD}_\text{DEF} \sqcup \text{CTOR}_\text{DEF}) \circ (\text{next}_\text{Sibling}^+ \circ \text{VARIABLE}_\text{DEF}) \right) \sqcup \left( (\text{METHOD}_\text{DEF} \sqcup \text{CTOR}_\text{DEF}) \circ ((\text{next}_\text{Sibling})^-)^+ \circ \text{VARIABLE}_\text{DEF} \right)$$

- a relation $\text{Procs2AccVars}$ which maps the nodes representing the methods on the nodes representing the variables that are accessed by the particular methods (local or instance):

$$\text{Procs2AccVars} = \left( (\text{METHOD}_\text{DEF} \sqcup \text{CTOR}_\text{DEF}) \circ \text{first}_\text{Child} \right) \circ \left( (\text{totalTree} \circ \text{dep: var}^- \circ \text{def}) \circ \text{VARIABLE}_\text{DEF} \right)$$

- a relation $\text{Procs2AccInstVars}$, based on the previous two relations, which maps the nodes representing the method on the nodes representing the instance variables that are accessed by the particular method. This relation can be constructed by intersecting the two relations just defined:

$$\text{Procs2AccInstVars} = \text{Procs2InstVars} \cap \text{Procs2AccVars}$$

- a relation $\text{InstVars2ProcsAcc}$ which maps the instance variables on the methods that access them. This relation can simple by constructed by taking the inverse of the last defined relation:

$$\text{InstVars2ProcsAcc} = (\text{Procs2AccInstVars})^{-1}$$

- a feature $\text{InstVars2NrofProcsAcc}$ which maps the nodes representing the instance variables on the number of methods accessing them:
InstVars2NrOfProcsAcc = countSecond(InstVars2ProcsAcc)

Composing the relation CLASS_DEF with the last defined feature in a suitable way results in another feature, which we call classdef2NrOfProcsAcc, containing pairs \((x, y)\) in which \(x\) represents the current class and \(y\) represents the number of methods that access a particular instance variable. From this point on, it is no longer interesting to know how many methods accessed a particular instance variable. According to the formula in Definition 3.3, we only need the average number of methods.

\[
\text{classdef2NrOfProcsAcc} = \\
(((CLASS_DEF \circ \text{first}_\text{child}) \circ \text{next}_\text{sibling}^+) \circ \text{OBJBLOCK} \circ \\
\text{first}_\text{child} \circ \text{next}_\text{sibling}^+ \circ \\
\text{VARIABLE_DEF} \circ \\
\text{InstVars2NrOfProcsAcc})
\]

averageAccInstVars = average(CLASS_DEF \circ \text{Procs2NrOfNotAccInstVars})

For defining the feature related to the term \(m\) (i.e. the number of procedures) we have to create a relation mapping the elements of the concept CLASS_DEF on the procedures that access instance variable and then pass this relation to the function countSecond to create the feature mapping these elements on the number of procedures being defined in the corresponding class. This feature will be called classdef2NrOfProcs. Creating this feature requires one new relation classdef2InstVars and one relation that has already been defined Procs2AccInstVars.

\[
\text{classdef2InstVars} = \\
(((CLASS_DEF \circ \text{first}_\text{child}) \circ \\
\text{next}_\text{sibling}^+) \circ \text{OBJBLOCK} \circ \\
\text{first}_\text{child} \circ \\
\text{next}_\text{sibling}^+ \circ \\
\text{VARIABLE_DEF} \\
\text{Procs2AccInstVars})
\]

\[
\text{classdef2Procs} = \text{classdef2InstVar} \circ (\text{Procs2AccInstVars})^-
\]

\[
\text{classdef2NrOfProcs} = \text{countSecond}(\text{classdef2Procs})
\]
At this point the features required to calculate the lack of cohesion are available. We have to define how to construct the features representing the numerator and denominator of the formula. Both features can easily be defined as the subtraction of two already defined features.

\[
\text{numerator} = \text{sub}(\text{averageAccInstVars}, \text{classdef2NrOfMeths})
\]
\[
\text{denominator} = \text{sub}(\text{classdef2One} \circ \text{classdef2NrOfMeths})
\]

### 3.4 Summary

At the beginning of this chapter we gave an outline of what kind of properties we would like to express using a formal language, namely a number of software metrics concerning the static structures of the program source. We stated that in order to measure particular properties, a particular model is needed from which we are able to map the program on a particular metric value. We selected four software metrics for which we gave the formal definitions and an explanation why these metrics are important in the process of software development. After that we have introduced the formalism that will be used for property expression, namely Description Logics. By giving a simple example, we showed how DL-languages can be used for expressing particular properties of models in general. After specifying how to extend DL-languages for providing more expression power, we pointed out how we represent graphs using this formalism. Since software metrics measure particular properties by assigning values to particular structures in the model, we introduced the concrete domain of the real numbers. In order to use this concrete domain in a suitable way, we have also introduced aggregated features and functions that are necessary in order to formulate the metrics as DL-expressions. The last part of this chapter discussed the DL-expressions of the metrics. It became clear that we are not able to calculate metric values for individuals of the domain. Using DL-expressions we are only able to construct new concepts. The result of a DL-expressing therefore is a set containing the individuals in the interpretation domain that satisfy the conditions in the DL-expression.
Model Construction

This chapter describes the abstraction from Program to Model as they appear in Figure 1.1. The class graph of the program, as defined in Chapter 2, will be the result of this process. Different steps are performed in order to construct the class graph. An overview of the steps to be taken is shown in Figure 4.1. Given the program-source, the abstract syntax tree is constructed, which will be used for creating the corresponding class graph. These two steps are performed by using different construction mechanisms. These mechanisms and the way they are applied will be discussed in this chapter.

![Figure 4.1: From source to class graph.](image)

The model construction process is performed using two tools: ANTLR and GROOVE [24]. Installation and configuration of these tools is discussed in Appendix A and B respectively.

4.1 Model Construction Mechanisms

Creating the abstract syntax tree (generally abbreviated to AST) from the source of a program is done by using a parser, generated by ANTLR. This is a command-line tool which requires a grammar specification as input and generates a parser implementing the specified
grammar. This grammar may contain actions for tree construction, which is supported by ANTLR. The parser generated by ANTLR can then be used for parsing source-files and constructing the abstract syntax tree.

We transform the AST into the corresponding class graph by applying graph transformations. The graph transformations are performed by using GROOVE. Since the abstract syntax tree is the starting point for this process, it must be in a format which can be imported in GROOVE. Therefore we have implemented some tree walkers (further referred to as tree visitors). Those visitors, basically, ‘walk’ through the AST, performing particular actions at each node in the tree. The visitors we have implemented create a similar structure using objects out of the GROOVE-library. The created structure is exported to a file that can be used for performing graph transformations.

### 4.1.1 Tree Visiting

Using the Java-parser generated by ANTLR results in the construction of the abstract syntax tree of the Java-file. When the abstract syntax tree is available it needs to be converted to a format in which we can later on reuse it for further processing. Tree visitors visit all nodes of the tree and perform specific actions. Visitors can be implemented using two approaches: depth-first or breath-first. In this project we have applied the depth-first approach. Since the abstract syntax tree is generated from a source-file, which always contains a finite number of characters, the corresponding abstract syntax tree is also a finite tree and therefore the depth-first approach nevers ends up in an infinite loop.

Converting the abstract syntax tree into a format that can later on be used as a start graph for the transformation process can be done in different ways. The most obvious visitor converts the abstract syntax tree into the identical structure in the target-format preserving parent-child relations. The problem using this approach is that it may no longer be possible to track the order of child nodes. This depends on the way of modelling. The data structure in which ANTLR models the abstract syntax tree provides means for visiting the child nodes in a predefined way, namely by using methods called `getFirstChild()` and `getNextSibling()`.

In graphs, however, there is no distinction between the different successors of a particular node: the successors are not ordered. Since the ordering of child nodes is crucial when performing graph transformations, we need an alternative approach: the binary tree. Converting the abstract syntax tree in a binary tree preserves the order of elements. In the binary tree every node has at most two outgoing edges. If it has no outgoing edges, the node in the original tree as a leaf-node. If it has one outgoing edge, the original node had at least one child but no siblings. If it has two outgoing edges, the original node had at least one child
and one sibling. Figure 4.2 shows the different structures and how the two visitors convert it to the target-format.

(i) Original tree  (ii) Identical tree  (iii) Binary tree

Figure 4.2: Converting AST-structure into identical or binary tree in Groove format.

4.1.2 Graph Transformation

Graph transformations are performed by applying graph transformation rules (also called graph production rules). First we will define what a graph production rule is and what we mean with “applying” such a rule [22].

Definition 4.1 (graph production rule) A graph production rule is a graph whose nodes and edges are partitioned into (possibly empty) sets $N^\text{use}$, $N^\text{del}$, $N^\text{new}$ and $N^\text{not}$ (respectively $E^\text{use}$, $E^\text{del}$, $E^\text{new}$ and $E^\text{not}$) such that

- $\text{src}(e) \in N^\text{del}$ or $\text{tgt}(e) \in N^\text{del}$ implies $e \in E^\text{del}$
- $\text{src}(e) \in N^\text{new}$ or $\text{tgt}(e) \in N^\text{new}$ implies $e \in E^\text{new}$
- $\text{src}(e) \in N^\text{not}$ or $\text{tgt}(e) \in N^\text{not}$ implies $e \in E^\text{not}$

An application of a graph production rule $P$, for which we define $P^{\text{pos}}$ as being the graph production rule equal to $P$ minus the NAC’s of $P$, to a given graph $G$ involves finding a total and surjective morphism $\mu^{\text{pos}} : P^{\text{pos}} \rightarrow Q^{\text{pos}}$ such that

- $\mu^{\text{pos}}(p) = \mu^{\text{pos}}(q)$ implies $p = q$ or $p, q \not\in N^\text{new}$ (i.e. $\mu$ is injective on the new nodes),
- $N_G \cap N_{Q^{\text{pos}}} = \mu^{\text{pos}}(N^\text{del} \cup N^\text{use})$ and $E_G \cap E_{Q^{\text{pos}}} = \mu^{\text{pos}}(E^\text{del} \cup E^\text{use})$ (i.e. the del and use nodes are projected by $\mu^{\text{pos}}$ into $G$ whereas the new nodes are kept disjoint from $G$),

$N^\text{not}$ and $E^\text{not}$ contain the elements that form negative application conditions (NAC’s) [13].
• $\mu^\text{pos}$ can not be extended to $\mu : P \rightarrow Q$, such that $N_P \setminus N_{P^\text{pos}} \subseteq N_G$ and $E_P \setminus E_{P^\text{pos}} \subseteq E_G$
(i.e. the NAC’s are present in $G$);

Given such a $\mu^\text{pos}$, $P^\text{pos}$ transforms $G$, the source graph, into a new graph $H$, the target graph, defined by

\begin{align*}
N_H &= (N_G \setminus \mu^\text{pos}(N^\text{del})) \cup \mu^\text{pos}(N^\text{new}) \\
E_H &= ((E_G \setminus \mu^\text{pos}(E^\text{del})) \cup \mu^\text{pos}(E^\text{new})) \mid N_H
\end{align*}

Graph production rules are often specified by two graphs $L$ and $R$, also called left-hand-side (LHS) and right-hand-side (RHS). $L$ contains the elements to be matched in the source graph, including the NAC’s. Every node $p \in N_L$, such that $p \notin N_R$, will be deleted from the source graph and every node $q \in N_R$, such that $q \notin N_L$ will be created in the target graph. The same holds for edges. We can apply this approach using Definition 4.1 by stating that

\begin{align*}
N_L &= N^\text{use} \cup N^\text{del} \\
E_L &= E^\text{use} \cup E^\text{del} \\
N_R &= N^\text{use} \cup N^\text{new} \\
E_R &= E^\text{use} \cup E^\text{new}
\end{align*}

The NAC’s ($N^\text{not}$ and $E^\text{not}$) are also part of $L$ in a specific way [13].

Specifying graph transformation rules is subject of a wide area of research [25]. During the last three decades, much research has been done on graph transformations in general. Two different approaches have been developed for specifying transformation rules, namely the single pushout approach (SPO) and the double pushout approach (DPO).

The main difference between the two approaches is that the DPO-approach adds two conditions for the application of a rule:

• **dangling conditions**: if a node $n$ is deleted, then the graph production rule *must* specify the deletion of the edges incident to $n$;

• **identification conditions**: every element that should be deleted in the source graph has only one pre-image in the LHS of the graph production rule.

Performing graph transformations requires a graph grammar to be specified.
Definition 4.2 (graph grammar) A graph grammar $G$ is a tuple $(G_0, P)$, where

- $G_0$ is the start graph of $G$,
- $P$ is the set of graph transformation rules (also called graph production rules)

Graph Production Rule Format

In this paragraph we briefly introduce the format of the graph production rules as we specify them in this report. In Appendix B a more detailed description of the syntax of the graphs, and especially graph production rules, is given. In this format we distinguish four different element types, which are exactly the four sets as stated in Definition 4.1 and listed in the respective order:

- **Reader** nodes and edges, represented by solid black arrows and blocks. These elements must exist in the source graph in order for the rule to be applicable. Applying the rule does not affect this elements. This type relates to the use element

- **Eraser** nodes and edges, represented by dashed blue arrows and blocks. These elements must exist in the source graph. After applying the rule, these elements will be deleted.

- **Creator** edges and nodes, represented by solid green arrows and blocks. These elements will be created after applying the rule.

- **Embargo** edges and nodes, represented by dashed red arrows and blocks. These elements represent the negative application conditions and applicability of the rule requires the absence of these elements.

The graphical representation of the rules is explained by using a simple example: a circular buffer with three slots. Figure 4.3 (i) shows how we model the circular buffer. Two operations that can be performed on the buffer are put and get, which are shown in Figure 4.3 (ii) and Figure 4.3 (iii), respectively.

When taking a look at the put-rule, it can be formulated in natural language as follows:

The put-rule requires an empty slot. After applying this rule, this first empty slot will be assigned an object and will become the last non-empty slot of the buffer.
Figure 4.3: Graph production rules that can be applied on the circular buffer.

This rule is formalised by a graph consisting of five nodes and five edges. Three nodes (one labelled ‘Buffer’ and two labelled ‘cell’) must exist in the source graph and will not be removed. There may not be a node labelled ‘Object’ that is connected to the node representing the first empty slot in the buffer. After applying this rule, a node labelled ‘Object’ will be created that is connected to the last element of the buffer by an edge labelled ‘val’. The edge labelled ‘last’ will then point to the last not empty slot of the buffer.

For the get-rule we do the same:

The get-rule requires the first slot of the buffer (pointed to by an edge labelled ‘first’) to have an object being assigned to it. After applying this rule, this object will be removed and the next slot will become the first slot of the buffer.

This rule is formalised by a graph consisting of four nodes and four edges. Three nodes (one labelled ‘Buffer’ and two labelled ‘cell’) need to exist in the source graph and will not be removed. One node labelled ‘Object’ must exist in the source graph and be connected to the first slot of the buffer by an edge labelled ‘first’. After applying this rule, this node, together with the edge pointing to it, will be removed. The edge labelled ‘first’ will then point to the next slot of the buffer, which may be filled or empty.
Applying the put-rule on the graph shown in Figure 4.3 (i) (the empty buffer) results in the graph shown in Figure 4.4.

![Figure 4.4: Circular buffer containing one object.](image)

### 4.2 AST-visitors

In this project we have implemented four visitors. They are shown in Figure 4.5 from which it becomes clear that they are structured using the Bridge Design Pattern [12]. The IdenticalVisitor converts the given tree to exactly the same structure in the target-format, the BinaryVisitor creates the binary tree of the given tree and the CombiVisitor combines these two approaches.

The BinaryDepVisitor we have implemented for the conversion to a suitable format does more than just converting the structure. Since we use GROOVE for performing the graph transformations, we have to cope with its limitations. One of those limitations is that GROOVE is not able to match nodes based on their labels. This is a problem when we want to look for places at which the same variable is being accessed or in cases when we want to track the procedure being called. The only way for solving this problem is to create a node at global scope. From other nodes having the same label an edge can be created pointing from that node to the node at global scope. All nodes having the same label will then point to the global node.

### 4.3 Graph Grammars

In this part of the report we discuss the transformation from abstract syntax tree to the corresponding class graph. The abstract syntax serves as an intermediate model between the program source and the corresponding class graph.
We specify three graph grammars that define the transformation of the abstract syntax tree. Those graph grammars add specific structures to the abstract syntax tree concerning

- control flow of procedures
- calling relations between procedures
- dependencies between statements

We enrich the original abstract syntax tree with static structure information concerning control flow, call and dependence information. For each structure we specify a separate graph grammar; they will be called $cflow$, $call$ and $dep$, respectively. Each graph grammar uses the structure of the abstract syntax tree and adds specific information by applying graph production rules. Since each graph grammar does not change the original abstract syntax tree structure, the order in which they are applied is not of any importance: the different graph grammars are confluent (see Figure 4.6). We do not prove this in a formal way. From the fact that each graph grammar does not change the original structure of the original abstract syntax and the fact that the information added by each graph grammar is not used by the others we infer that it is very likely for the three graph grammars to be confluent.

We discuss the three graph grammars based on the assumption that the abstract syntax tree is the initial graph. From the confluence-characteristic of the three graph grammars we infer that this is valid, since the final graph of each graph grammar still contains the original structure of the abstract syntax tree. For each graph grammar we first describe the basic...
4.3 Graph Grammars

Figure 4.6: Transforming graph $G$ into graph $H$ by applying three confluent graph grammars $cflow$, $call$ and $dep$.

mechanisms it relies on. Then, a number of issues that arose when creating that particular graph grammar will be discussed and finally some specific graph production rules will be shown and explained. The total set of graph production rules of each graph grammar is shown in Appendix E.

The labels of edges do not have to be simple labels. They may also be regular expressions based on simple labels. The advantage of regular expressions is that when specifying graph production rules using regular expressions we do not always need to specify all nodes and edges to be matched. Using regular expressions as labels of edges from one node to another node we are able to capture multiple cases by specifying one single rule. Regular expressions are, for example, very useful in cases you want to perform a transformation on a sequence of elements all being connected to each other with equal-labelled edges. The graph grammars we will discuss use regular expressions. The syntax of regular expressions is explained in Appendix E.

4.3.1 Control flow structure

The graph grammar, which specifies how to add control flow structure to the abstract syntax tree should construct the control flow graph of every procedure that is defined in the class that is represented by the resulting class graph. Since each procedure has its own control flow graph, we have to relate the control flow graph to the procedure it belongs to. The
root-node of the procedure (the node labelled ‘METHOD_DEF’ in case of a method and the node labelled ‘CTOR_DEF’ in case of a constructor) is the most obvious node to be used for this goal. After constructing the control flow graph of a particular procedure, the root-node will have two additional outgoing edges: one edge labelled ‘cflow-begin’ connecting the root-node with the start node and one edge labelled ‘cflow-end’ connecting the root-node with the end node of the control flow graph.

Adding control flow information to the abstract syntax tree can be done by using two different approaches: *bottom-up* or *top-down*. The bottom-up approach starts performing graph transformations at the leaves of the abstract syntax tree. The resulting graph should be the control flow graph corresponding to the analyzed (part of the) abstract syntax tree. The transformation can also be performed by applying the top-down approach. This means that it starts at the root of the abstract syntax tree (or at least at the root of the part of the abstract syntax tree that will be transformed). Applying both approaches should result in the same control flow graph, otherwise the specified transformation rules are not correct.

Independent of the approach that is being applied, two operations are defined for creating (control) flow graphs: *nesting* and *sequencing*.

**Definition 4.3** If $F_1 = (G_1, a_1, z_1)$ and $F_2 = (G_2, a_2, z_2)$ are flowgraphs, then the sequence $F_1; F_2$ of $F_1$ and $F_2$ is the flowgraph $(G_1; G_2, a_1, z_2)$ where $G_1; G_2$ is the directed graph which is obtained from the union of $G_1$ and $G_2$ by identifying the nodes $z_1$ and $a_2$.

**Definition 4.4** If $F_1 = (G_1, a_1, z_1)$ and $F_2 = (G_2, a_2, z_2)$ are flowgraphs and $x$ is a node of $G_1$ with out-degree 1 (called a procedure node) then the nesting $F_1(F_2$ on $x)$ is the flowgraph $(G_1(G_2$ on $x), a_1, z_1)$ where $G_1(G_2$ on $x)$ is the directed graph which is obtained from the union of $G_1$ and $G_2$ by deleting the edge whose source is $x$, identifying $x$ and $a_2$ and identifying $z_2$ and the successor of $x$.

Control flow graphs can be constructed by performing these operations on existing control flowgraphs using elementary ones, so called *prime graphs*, being flow graph that cannot be decomposed non-trivially by sequencing and nesting [10]. Figure 4.7 gives an overview of the prime graphs that are used for constructing more complex control flowgraphs.

Applying either of the two different approaches relies on the fact that at some point in the transformation process, a particular part of a control flowgraph is replaced by a more complex structure. When using the bottom-up approach, the structure to be inserted is already known; in the case of the top-down approach, the replacing structure will be constructed later. The basis is that when finding a single expression, we add a procedure-node to the
control flow graph (in fact the node itself is already present, it is only needed to add a self-edge labelled ‘PROC’ to it); when matching a conditional expression (i.e. an if-, for- or while-statement), we need to add a predicate-node together with two outgoing edges and their target-nodes, one for the true- and one for the false-branch.

A comparison between the two approaches has led to the decision to use the top-down approach instead of the bottom-up approach. The reason for this is that applying the sequence or nesting operation requires pairs of nodes to be identified. Take for example the sequence of two graphs $G$ and $H$. The resulting graph $F = G; H$ is obtained by identifying the end node $i$ of $G$ and the start node $j$ of $H$. The problem with performing this operation is that the in-degree of node $i$ and the out-degree of node $j$ are unpredictable. Working around this problem can be done by creating a new edge between the two nodes instead of identifying them. Another alternative for resolving this problem is to introduce an extra end-node to every flowgraph as a *successor* of the regular end-node of that flowgraph and an extra start-node as an *ancestor* of the regular start-node of that flowgraph. Using this alternative it is always assured that the additional end-node of the first flowgraph always has in-degree one and that the additional start-node of the second flowgraph has out-degree one. The objection to this alternative is that we need to add rules to the graph grammars which take care of deleting the elements that are not part of the actual control flow graphs.

When taking a short look at this graph grammar, there are two graph production rules that
are useful to explain: one rule that triggers the rest of the graph transformation process and another rule that performs the nesting or sequence operation. The second rule is just a representative of all other rules, since they all perform likewise transformations.

![Graph Production Rules](image)

**Figure 4.8:** Graph production rules of the control flow GPS.

First of all, we emphasise that the transformation process needs to start but also needs to stop at a certain point. The idea is as that before an expression (or a block of expressions) can be included in the control flowgraph, the root-node of that expression must have two temporary outgoing edges: one labelled `embed-begin` and another labelled `embed-end`.

The edges labelled `embed-begin` and `embed-end` point, respectively, to the start-node and end-node of the control flowgraph of the expression. If those edges are present, we can apply a graph production rule which performs the nesting or sequence operation on the subtree of the root-node of the expression. The graph production rule will then remove those edges, indicating that the transformation process is one step further. This prevents from entering a infinite loop in which graph production rules are continuously applicable.

Figure 4.8 (i) shows the graph production rule triggering the control flowgraph creation for a single method. This rule contains one negative application condition stating that the method-root node may not yet have outgoing edges labelled `cflow-begin` or `cflow-end`, because that would mean that the control flowgraph of this method has already been created or is still under construction. If this rule is applied, the node labelled `BLOCK`, representing the root of the block of statements of that method, will get two new outgoing edges labelled `embed-begin` and `embed-end`. The rule in Figure 4.8 (ii) takes care of the continuation of
the transformation process by adding the embed-edges to the first statement in that block.

4.3.2 Call structure

Since call graphs represent calling relations between procedures, the structure to be added to the abstract syntax tree consists of edges between nodes representing the procedures. There exists a calling relation\(^1\) between procedures \(a\) and \(b\) if \(a\) has at least one statement in which \(b\) is called. The basic idea of this graph grammar is that we will look for nodes labelled ‘METHOD_DEF’ or ‘CTOR_DEF’ having a subtree containing at least one node labelled ‘METHOD_CALL’ or a node representing a constructor call. The first step we perform is actually not needed, but it increases the performance during the rest of the transformation process. The graph transformation rule concerning this step is shown in Figure 4.9 (i) and will be explained in one of the coming paragraphs. The nodes labelled ‘METHOD_CALL’ are already indirectly related to the procedure being called. This enables us to match the root-node of the procedure being called and therefore we are able to create an edge from the root-node of the procedure having the procedure call in its subtree to the root-node of the procedure being called.

Determining which procedure is called in a particular statement is based on matching the names of the called procedure and the defined procedures. A better way to do this is by matching procedure signatures. A procedure signature captures information that uniquely describe the properties of a procedure, such as its name, its return type and its parameters. The problem in the approach we apply (performing graph transformations using unattributed graph grammars) is that enabling signature matching is not easy to achieve. One way to enable signature matching is by adding elements to the class graph specifying the signature of the procedures being defined or called. When applying attributed graph grammars, signature matching seems to be much easier.

As already mentioned in Chapter 2 we represent calling relations between methods and constructors. Each of these call categories need their own graph production rules. Therefore, we show, and briefly explain, a number of rules dealing with these different concepts.

Before we are able to create the edges representing the calling relations between methods and constructors, we need two rules that trigger the other rules. One of these rules is shown in Figure 4.9 (i), the other rule has an analogue structure. As we have already pointed out in this chapter, due to limitations of GROOVE, the AST-visitor adds global nodes to the abstract syntax tree which enables ‘label-matching’ of nodes. This rule matches the node

\(^1\)Recall that the calling relation is directed: a calling relation from procedure \(a\) to procedure \(b\) is not the same as a calling relation from \(b\) to \(a\).
Figure 4.9: Graph production rules of the call GPS.
4.3 Graph Grammars

labelled ‘METHOD_DEF’, the node labelled with the name of the method (which is the ancestor of the node labelled ‘PARAMETERS’) and the global node. After applying this rule, there will be an edge from the root-node of the method to the global node. Adding call information could also be done without this rule. Using this rule, however, increases the performance of the graph transformation process, since the rules which will hereafter add the actual call information need to match less nodes than they would have to if this rule was not present.

The graph production rule shown in Figure 4.9(ii) is applicable if there exists a method which contains an expression in which another method is called (or in which the same method is called recursively) and there does not yet exist an edge from the calling method to the called method labelled ‘call:calls’, representing the calling relation. In those cases, a new edge will be created pointing from the node labelled ‘METHOD_DEF’, representing the calling method, to another node with the same label, representing the called method.

The third production rule, shown in Figure 4.9(iii), describes what should be done if a particular method contains a statement calling a method in some library-class. As already mentioned in this chapter, these calls are part of the call graph, but are a little different from the calls to internal methods. Therefore the edges representing external calls are labelled ‘call:external-calls’ instead of ‘call:calls’.

Figure 4.9(iv) shows the handling of methods calling a constructor of the class itself. This rule is applicable if there exists a node in the subtree of the method-root and a constructor-root node pointing to the same global node.

In each of the production rules there is a negative application condition checking that there does not yet exist an edge from the current method-root node to the node representing the called method.

4.3.3 Dependence structure

The dependence information to be added consists of edges between the nodes representing variable-access and the node representing the variable-declaration. These variables can be parameters of the corresponding procedure, locally defined variables, instance variables of the modelled class or instance variables of other classes. The definition of the metric concerning dependence structure, the lack of cohesion in procedure, only deals with the access of instance variables by the procedures of the class itself. Therefore, when a statement accesses a variable, it is sufficient to know whether this variable is an instance variable or not. This information can be extracted from the dependence structure, which will be added to the abstract syntax tree.
The abstract syntax tree contains a subtree for every variable- or parameter-definition. The subtree of a variable-definition may be nested in the class graph at class-level or at procedure-level. In the former case, it is the definition of an instance variable, in the latter case, it refers to the definition of a local variable. It is clear that parameter-definitions are nested in the class graph at procedure-level.

The graph grammar which takes care of adding dependence information to the abstract syntax tree contains only three rules: one rule for handling instance variables, one for handling local variables and one for handling parameters. The first two rules shown in Figure 4.10 will be explained.

The rule shown in Figure 4.10 (i) and (ii) describe how dependence information concerning, respectively, instance variable and local variable access is added to the model. When comparing the two rules, it is clear that the latter requires a node labelled ‘METHOD_DEF’ to be matched, indicating that the variable being accessed is a local variable. In the former rule the node representing the variable declaration and the node representing the method in which the instance variable is accessed, need to be siblings. This becomes clear from the labels of the edges from the node these nodes and the node labelled ‘OBJBLOCK’: both edges have the label ‘first_child.next_sibling*’.

(i) instance-var.gpr  
(ii) local-var.gpr

Figure 4.10: Graph production rules of the dependence GPS.

The three rules can not be part of the same graph grammar. The reason for this is that when a particular statement accesses a variable, we first have to check whether this is a local variable or a parameter. This is the case if there exists a node representing a local variable or a parameter declaration which has a relation to the same global node labelled with the name of the variable as the node representing the variable-access. If there does not exist such a node, it refers to an instance variable.
The reason for putting these rules in different graph grammars is that applying one of the rules removes the relation between the node representing the variable-access and the global node. After removing this relation both graph production rules are not applicable anymore. In the case a local variable is accessed which has the same name as one of the instance variables, both rules are applicable. In these cases the rule concerning local variables has priority over the rule concerning instance variables. GROOVE does not provide means for prioritizing graph production rules. Therefore, the only way of solving this problem is to put both rules in two distinct graph grammars and first apply the graph grammar containing the local variable-rule.

4.4 Summary

In this chapter we have explained how the class graph is constructed. We have discussed two construction mechanisms that have been applied, namely tree visiting and graph transformations. We discussed that, since we use the tool GROOVE for performing the graph transformation, the tree visitor used in this project, the \texttt{BinaryDepVisitor}, has to do more than only the conversion from the original format to the format supported by GROOVE, because GROOVE does not provide functionality for label-matching. Transforming the abstract syntax tree into the class graph is then performed by three confluent graph grammars: one graph grammar for adding control flow information, one for adding call information and one (actually two) for adding dependence information. For each graph grammar we have described the basic mechanism and explained a small number of production rules.
Software Metric Calculation

The final goal of this project is to express the selected metrics as properties of the class graph using Description Logics. In order to get the result of a DL-expression a number of steps need to be performed. First of all, we need to define what the results mean; does the evaluator to be implemented for example say ‘yes’ or ‘no’ or does it return a set of elements or just a single element. In Chapter 3 we have already referred to the Description Logic grammar listed in Appendix C which specifies how to define new concepts and relations given existing ones and using the constructors to combine them. We gave an overview of how to use concept and relation constructors and explained why and how Description Logics are extended with concrete domains.

In this chapter we explain how the result of DL-expressions is determined. In order to be sure that the DL-expressions are evaluated correctly, we compare the results with values as being determined for the same metrics by other tools like Borland Together and Prometrix.

### 5.1 Expression Evaluation

Every DL-expression defines a new concept. That means that the result of a DL-expression is a set of individuals which satisfy the conditions as stated in the DL-expression.

At first sight, there are two approaches for evaluating DL-expressions.

- matching the characteristic graph
- applying the algebraic semantics

The first approach is based on the construction of a graph that reflects the given DL-expression. Since the name of relations and concepts in the DL-expression relate to the
labels of edges (self-edges in the case of concepts) in the model to be matched, we should be able to create a graph that contains particular nodes and labelled edges, together representing the DL-expression. When this characteristic graph is available, it can be matched against the underlying model. From the matchings we can track which individuals satisfy the DL-expression.

The second approach applies the algebraic semantics of the constructors as defined in Chapter 3. We create a graph related to every atomic concept and relation as they appear in the DL-expression, match these small graphs against the underlying model resulting in a collection of matching elements and then combine those collections according to the semantics of the applied constructors.

In this project we have applied the approach based on the application of the algebraic semantics. This is done because it is difficult to specify how to construct the characteristic graph if a DL-expression contains for example a number restriction constructor or aggregate features or even functions.

### 5.1.1 Collection construction

When parsing a DL-expression its unique abstract syntax tree is created. Given the abstract syntax tree of a DL-expression, the evaluator constructs the corresponding set of individuals (this can be interpreted as a new defined concept using existing concepts and relations and the available constructors).

As we have already mentioned, concepts in a graph are represented by the labels of self-edges (except for the self-edges labelled `call:calls` representing a recursive calling relation of a method). Constructing the collection of individuals of a particular concept is performed in several steps. First, we construct a graph $G_C$ consisting of only one node labelled with concept-name $C$. Then we look for the matchings of this graph in the model. The nodes of the underlying model matching $G_C$ will be the elements of the returned collection. In the case of the construction of the collection containing relations, we make a graph $G_R$ consisting of two distinct nodes being related by an edge labelled $R$. In this case, the returned collection contains the edges that match $G_R$. Since matching the edges is performed for every relation separately, the name of the relation (the labels of the edges) is not important anymore after the matching. When composing two collections representing two relations, we do not need to know the name of the relation themselves; having the pairs of nodes belonging to a particular relation is sufficient. Therefore, after the matching-part of edges we only store the identities of the nodes together with their role, with respect to the original edge (i.e. if it was the source or target of the edge).
5.2 Running the tool

The collection of a composed concept or relation is constructed by walking the tree using the depth-first visiting algorithm. When the visitor enters a node representing a constructor, it first constructs the collections of its parameters and then applies the algebraic semantics of the constructor. For example, when the visitor enters a node of type AND it knows that this node has two children. The collections belonging to these children will be constructed after which the intersection of the two collections is constructed.

In this section we only discuss the abstract syntax tree of the fan-out metric and show what steps need to be taken to construct the resulting collection. The abstract syntax trees of the DL-expressions of the other metrics are shown in Appendix F.

Figure 5.1 shows the abstract syntax tree of the DL-expression that returns all the nodes representing the procedures having a fan-out greater than 1.

From this abstract syntax tree we see that this metric is based on the relation call:calls and on the concepts METHOD_DEF and CTOR_DEF. We apply the function countSecond on the relation call:calls for creating the set \( S_{\text{fanout}} \) containing pairs \((x_1,y_1)\), in which \(x_1\) represents a procedure and \(y_1\) is the number of procedures called by \(x_1\). In order to filter the procedures having a fan-out greater than 1 out of \( S_{\text{fanout}} \), we need to construct the set \( S_{\text{filter}} \) containing pairs \((x_2,1)\) for each procedure represented by \(x_2\). By applying the predicate \( > \) (greater than), we only select the nodes \(n\) which are represented in \( S_{\text{fanout}} \) by a pair \((n,v)\) such that \(v > 1\).

5.2 Running the tool

The result of the DL-expressions is a collection of nodes. The elements in the collection of the fan-in, fan-out and cyclomatic complexity metric refer to procedures satisfying the
DL-expression. The lack of cohesion should return the node representing the class being modelled if it satisfies the conditions stated in the DL-expression. In the former case, we want to know which procedures satisfy the conditions. In that case we are interested in the names of the methods that are part of the resulting collection. In order to get the name, we have to go back to the underlying model. In Chapter 2 we have shown that the Java-grammar can be taken as a reference point for determining the structure of the abstract syntax tree (the type graph). The field-rule of the Java-grammar states that the third child of the nodes labelled ‘METHOD_DEF’ carries the name of methods. When we have the node labelled ‘METHOD_DEF’ we have to take three edges (one time the first_child-edge and two times the next_sibling-edge) to get to the node labelled with the name of the method. In the latter case, the tool will indicate whether the methods in the given class satisfy the condition concerning the lack of cohesion. Since we know what class we are dealing with, it is not really interesting to get the name of the class.

In the next paragraphs we will discuss the results of the tool for the different metrics. The model on which we run the different expressions is stored in the file Calculator.gst. This model is the class graph of the class Calculator from the running example. The notation used for the DL-expressions is the same as shown in Chapter 3, because this makes the expressions more readable. The syntax of the DL-language used in this project is shown in Appendix D.

Because we continuously use the name procedure to refer to both methods and constructors, we will again use the concept called Procs as we have already defined in Chapter 3.

\[
\text{Procs} = \text{METHOD_DEF} \sqcup \text{CTOR_DEF}
\]

We will discuss the results of each metric. Since a DL-expression returns the set of individuals that satisfy the conditions we are, for example, not able to get the fan-in of a particular method by giving the node representing it. We can only construct the set of individuals that have a particular value for one of the metrics. When discussing the fan-in and fan-out metrics we will show all the steps to be taken. For the other metrics we will only show the results.

5.2.1 Procedures fan-in and fan-out

For determining the fan-in or fan-out of procedures, we start the tool with the following command:

\[
$ \text{java Evaluator -sg Calculator.gst -l procedure}$
\]
5.2 Running the tool

The tool now asks for the DL-expression by printing:

```$ Enter DL-expression:$```

Assume that we are interested in the procedures having a fan-in greater than 0. In order to get those procedures we have to enter the following DL-expression:

```$ >(countFirst(call:calls), mapToConstant(Procs, 0))$```

The output of the evaluator is shown in Figure 5.2.

![Figure 5.2: Procedures having a fan-in metric value greater than 0.](image)

In order to be able to compare the results with those of other tools we are not interested in whether this metric value exceeds a certain value. We need to know the value for every procedure exactly. Therefore, we use a slightly different DL-expression. Instead of using the predicate-symbol ‘>’ (greater than), we use the ‘==’-symbol (equal to). For getting the names of the procedures with a fan-in of $n$, we use the DL-expression shown below.

```$ ==(countFirst(call:calls), mapToConstant(Procs, n))$
$ ==(countSecond(call:calls), mapToConstant(Procs, n))$```

The result of these metrics for the different procedures as being determined by the tool we have implemented, are shown in Table 5.1.

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Fan-in</th>
<th>Fan-out</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Calculator</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>main</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>mul</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>setA</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>setB</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>sub</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 5.1: The fan-in and fan-out for each procedure.
5.2.2 Cyclomatic complexity

We enter the following DL-expression, which should return all nodes representing the procedures that have a cyclomatic complexity greater than 1 (see Chapter 3):

\[ \text{add(second(Procs2Preds),} \]
\[ \text{mapToConstant(Procs, 1)),} \]
\[ \text{mapToConstant(Procs, n))} \]

in which Procs2Preds represents the relation which maps each node representing a procedure on the nodes representing the predicate nodes of the control flow graph of that procedure. The results are shown in Table 5.2.

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Cyclomatic complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>1</td>
</tr>
<tr>
<td>Calculator</td>
<td>1</td>
</tr>
<tr>
<td>main</td>
<td>5</td>
</tr>
<tr>
<td>mul</td>
<td>1</td>
</tr>
<tr>
<td>setA</td>
<td>1</td>
</tr>
<tr>
<td>setB</td>
<td>1</td>
</tr>
<tr>
<td>sub</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 5.2: The cyclomatic complexity for each procedure.

5.2.3 Lack of Cohesion

As we have already pointed out in Chapter 3, we are not able to give one DL-expression for constructing the set of elements that fulfill particular conditions concerning the lack of cohesion. We can only determine the values that are needed to calculate it and then perform the calculation by hand.

Table 5.3 gives an overview of how many methods access the different instance variables.

<table>
<thead>
<tr>
<th>Instance variable</th>
<th>Number of procedures</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>4</td>
</tr>
<tr>
<td>b</td>
<td>4</td>
</tr>
<tr>
<td>result</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 5.3: The number of methods that access particular instance variables.
The average number of method accessing a particular instance variable is 4. The number of procedures that access instance variables is 6. For determining the value of the numerator, we use the following DL-expression:

\[ n = \text{sub(averageAccInstVars, classdef2NrOfMeths)}, n \]

This results in the value -2 (= 4 - 6). The value of the denominator is determined by using the DL-expression shown below.

\[ n = \text{sub(classdef2One, classdef2NrOfProcs)}, n \]

This results in the value -5. The lack of cohesion is defined as the their division: \[ \frac{-2}{-5} = 0.40. \]

### 5.2.4 Comparison

To be sure that the DL-expression evaluator gives a correct result, we use other software development tools, which support metric value calculation, to compare the results. Prometrix [21] will serve as a reference point for comparing our results concerning the fan-in and fan-out and the cyclomatic complexity of several procedures. We use Borland Together [31] for checking our results concerning the lack of cohesion and the cyclomatic complexity metric.

#### Procedures fan-in and fan-out

Before being able to use Prometrix for checking our results, we have to translate the program into an equivalent program in the programming language Pascal. The pseudo Pascal-source of the program that served as input for Prometrix is listed in Appendix G.

Figure 5.3 shows the results of Prometrix concerning the fan-in and fan-out of the original main-method. This figure states that the fan-out of the main-method is 10. We had determined that it would have a fan-out of 4 (see Table 5.1). The reason for this difference is because Prometrix also count the calls to libraries if that option is enabled. Since the main-method calls six library procedures (BufferedReader, InputStreamReader, println, readLine, equals and parseInt), the value we determined is correct. The value 0 for the fan-in of this method is equal to the value we determined.

Figure 5.4 shows what values are determined by Prometrix for the Calculator-constructor in the original Java-source. This figures states that the fan-in and the fan-out of this procedure are 2 and 1, respectively. The results of our tool agree with this.
Figure 5.3: The results concerning fan-in and fan-out of the `main`-procedure as generated by Prometrix.

<table>
<thead>
<tr>
<th>Module</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>main</code></td>
<td></td>
</tr>
<tr>
<td>Size</td>
<td>21 nodes</td>
</tr>
<tr>
<td>Fan-in</td>
<td>0</td>
</tr>
<tr>
<td>Fan-out</td>
<td>10</td>
</tr>
</tbody>
</table>

Figure 5.4: The results concerning fan-in and fan-out of the `calculator`-procedure as generated by Prometrix.

<table>
<thead>
<tr>
<th>Module</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>calculator</code></td>
<td></td>
</tr>
<tr>
<td>Size</td>
<td>3 nodes</td>
</tr>
<tr>
<td>Fan-in</td>
<td>1</td>
</tr>
<tr>
<td>Fan-out</td>
<td>2</td>
</tr>
</tbody>
</table>

**Cyclomatic complexity**

Figure 5.5 gives an overview of the values for the cyclomatic complexity of each procedure as generated by Borland Together. Figure 5.6 and 5.7 show the results for the `main`- and `sub`-method, respectively, as generated by Prometrix. When comparing these results with those shown in Table 5.2, we see that they agree.

<table>
<thead>
<tr>
<th>Item</th>
<th>CC</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;default&gt;</code></td>
<td>5</td>
</tr>
<tr>
<td><code>Calculator</code></td>
<td>5</td>
</tr>
<tr>
<td><code>Calculator</code></td>
<td>1</td>
</tr>
<tr>
<td><code>add</code></td>
<td>1</td>
</tr>
<tr>
<td><code>main</code></td>
<td>5</td>
</tr>
<tr>
<td><code>mul</code></td>
<td>1</td>
</tr>
<tr>
<td><code>setA</code></td>
<td>1</td>
</tr>
<tr>
<td><code>setB</code></td>
<td>1</td>
</tr>
<tr>
<td><code>sub</code></td>
<td>2</td>
</tr>
</tbody>
</table>

Figure 5.5: The results concerning the cyclomatic complexity as generated by Borland Together.
5.3 Summary

Table 5.1: Overall Structure Metrics

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>McCabe’s metric</td>
<td>5</td>
</tr>
<tr>
<td>Prather’s metric</td>
<td>36</td>
</tr>
<tr>
<td>Lambda metric</td>
<td>28.00</td>
</tr>
<tr>
<td>Basili-Hutchens SYNC</td>
<td>22.08</td>
</tr>
<tr>
<td>Sum VINAP</td>
<td>35</td>
</tr>
<tr>
<td>Product VINAP</td>
<td>29</td>
</tr>
<tr>
<td>Extended product VINAP</td>
<td>35</td>
</tr>
</tbody>
</table>

Figure 5.6: The results concerning overall structure metrics of the procedure `main` as generated by Prometrix.

Table 5.2: Overall Structure Metrics

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>McCabe’s metric</td>
<td>2</td>
</tr>
<tr>
<td>Prather’s metric</td>
<td>2</td>
</tr>
<tr>
<td>Lambda metric</td>
<td>2.00</td>
</tr>
<tr>
<td>Basili-Hutchens SYNC</td>
<td>2.00</td>
</tr>
<tr>
<td>Sum VINAP</td>
<td>3</td>
</tr>
<tr>
<td>Product VINAP</td>
<td>2</td>
</tr>
<tr>
<td>Extended product VINAP</td>
<td>3</td>
</tr>
</tbody>
</table>

Figure 5.7: The results concerning overall structure metrics of the procedure `sub` as generated by Prometrix.

Lack of cohesion

The formula we used for calculating the lack of cohesion is also implemented in Borland Together. The only difference is that the formula used by Borland Together multiplies the result of the division with 100, to get a percentage. The value Borland Together returns for the lack of cohesion is shown in Figure 5.8 namely 40. When we multiply our result with 100, we get the same value.

Figure 5.8: The results concerning the lack of cohesion as generated by Borland Together.

5.3 Summary

We have shown how DL-expressions are evaluated. Two approaches have been mentioned to do this: creating the characteristic graph or applying the algebraic semantics. Thereafter, we showed how to use the tool and what it returns. Finally, we compared the results of
the DL-expression evaluator we have implemented with results generated by other software development tools. It turned out that the results of our tool with respect to the fan-in, fan-out and cyclomatic complexity were correct. Our value for the lack of cohesion in methods was also correct.
Discussion and Conclusion

In this project, we aimed at developing a technique for modelling a specific software artifact, namely the source code, for which we want to express properties using a formal language. The model we defined, called class graph, is a graph which models a selection of static structures of the program source in an object-oriented programming language.

Software metrics related to the static structures are expressed as properties of class graphs. We have selected a set of metrics, namely the method fan-in and fan-out, the cyclomatic complexity and the lack of cohesion in methods, which we expressed in a formal language called Description Logics.

We have constructed the class graph by taking the abstract syntax tree of a single Java-class. This abstract syntax tree, converted into a suitable format, served as the start graph for the graph transformation process which resulted in the class graph. The graph transformation process is performed by using the tool GROOVE.

At the end of this project we implemented an DL-expression evaluator which enabled us to check whether particular elements of the class graph of the running example satisfied a number of properties. We have seen that evaluating the DL-expressions resulted in the correct values for all four metrics.

6.1 Evaluation

The way we defined class graphs may not be totally satisfactory. Specific parts of a class graph represent control flow structure of procedures, but it does not support the control flow of language constructions, such as switch, try-catch and break-continue statements. Such language constructions require a alternative representation of control flow, especially
in the case of handling try-catch statements. There exist many different types of exceptions. Modelling exceptions requires a different way of modelling control flow, which preserves the reason of exceptions. This is currently not supported since we model control flow by single edges with a fixed label representing the sequencial relation between statements.

The graph grammars we defined do not support assigning attributes to nodes or edges. We explained that when adding call information to the abstract syntax tree, attributed graph grammars may be very useful, since it also supports element matching based on their attributes. Another reason for using attributed graph grammars may be the fact that it provides means to define metrics as attributes of elements.

We have experienced that expressing the selected metrics in a particular DL-language requires several extensions. Even extending the DL-language with concrete domains and aggregate functions did not provide enough expression power. The main reason for this is that the calculation of software metrics requires arithmetic operations to be performed. This requires the introduction of new functions. From this perspective we can conclude that the decision to use Description Logic was somewhat inappropriate.

We have pointed out that there are two basic approaches for evaluating DL-expressions. In this project we chose the approach which applies the algebraic semantics. An alternative approach is based on the construction of the characteristic graph of DL-expressions. When comparing these approaches from a performance perspective, we think that the alternative approach is preferable, since it requires the matching of only one graph, the characteristic graph. When applying the algebraic semantics, we first need to construct a graph for every concept and relation and match these graphs for getting the collection of matching elements in which case the number of matchings to be applied is equal to the number of concepts and relations the DL-expression refers to.

### 6.2 Related Work

Dori developed the Object Process Methodology [9]. The relation with this project is that OPM has been introduced to describe one model which encapsulated three major aspects in software systems, namely *function*, *structure* and *behaviour* just as we introduced one concept, class graphs, for modelling three kinds of static structure of program sources. The difference between them is that the OPM is used to *prescribe* the functionality and structure of the system.

The work of Baroni described in [6] relates to this project in the sense that she formalized object-oriented software metrics too. The difference is that she used the Object Constraint
6.3 Future Work

Language to formalize the MOOSE metric suite introduced by Chidamber and Kemerer.

Concerning the application of graph grammars for transforming one model into another model, Mens has done comparable efforts. He applied graph transformation for refactoring [18]. He introduced a graph presentation of aspects that should be preserved by a refactoring. He also worked on a project which aimed at developing a graph based meta model for object-oriented software metrics. In [19] he describes how type graphs are used as a metamodel to define a set of generic metrics which allows the expression of a large number of object-oriented metrics in a general, flexible and extensible way.

Since the widely applied pair graph grammars are restricted to context-free productions and one-to-one correspondences between objects in related data structures, Schürr introduced Triple Graph Grammars [27] for compensating these deficiencies. Triple graph grammars extend the original pair graph grammar approach to the case of context-sensitive productions and offer separate correspondence rules and graphs for modelling m-to-n relationships between related graphs.

6.3 Future Work

It is needed to revise the Description Logic grammar specification and study the possibilities for making it more generic. Implementing the DL-expression evaluator has been done in an ad-hoc way. In the future, this implementation must be systematically designed and implemented in order to be for example more easily adaptable, extensible and maintainable.

In the future it is worth studying the opportunities of using attributed graph grammars, since they simplify the way of expressing software metrics as properties of the class graph. Metrics can then be added to the class graph as attributes of particular elements. They may also enable the concept of procedure signature matching which is required when a program contains multiple procedures with the same name, but different signatures. AGG [30] is a tool for performing graph transformations using attributed graph grammars. This tool may then be very useful.
Bibliography


Harmen Kastenberg 81
Appendix A

ANTLR

A.1 Installing ANTLR

To install ANTLR, download the ZIP-file of the latest ANTLR version from http://www.antlr.org/download.html. Unpack the ZIP-file to some convenient directory (call this directory ANTLR-HOME).

A.2 Running ANTLR

It is important that you set the PATH and CLASSPATH environment variables correctly. You can work around this issue by passing the correct classpath as an additional parameter to the java-commands by running the following commands on a Windows-box:

1 $ cd GRAMMAR-HOME

2 $ set PATH=%PATH%;JDK-HOME\bin

3 $ java -cp ANTLR-HOME\antlr-2.7.2\antlr.jar;. antlr.Tool expr.g

Alternatively, you can add two environment variables as User variables, namely PATH and CLASSPATH, and give them the values as specified under System variables and add JDK-HOME to the PATH- and ANTLR-HOME to the CLASSPATH-variable.

1 GRAMMAR-HOME is the directory in which you placed the grammar-file
2 JDK-HOME is the home-directory of the java-compiler, for example C:\Program Files\j2sdk1.4.1.02
3 expr.g is the file in which you specify the grammar
ANTLR Appendix A.

A.3 ANTLR Java 1.4 Grammar

class JavaRecognizer extends Parser;
options {
    k = 2; // two token lookahead
    exportVocab=Java; // Call its vocabulary "Java"
    codeGenMakeSwitchThreshold = 2; // Some optimizations
    codeGenBitsetTestThreshold = 3;
    defaultErrorHandler = false; // Don't generate parser error handlers
    buildAST = true;
}

tokens {
    BLOCK; MODIFIERS; OBJBLOCK; SLIST; CTOR_DEF; METHOD_DEF; VARIABLE_DEF;
    INSTANCE_INIT; STATIC_INIT; TYPE; CLASS_DEF; INTERFACE_DEF;
    PACKAGE_DEF; ARRAY_DECLARATOR; EXTENDS_CLAUSE; IMPLEMENTS_CLAUSE;
    PARAMETERS; PARAMETER_DEF; LABELED_STAT; TYPECAST; INDEX_OP;
    POST_INC; POST_DEC; METHOD_CALL; EXPR; ARRAY_INIT;
    IMPORT; UNARY_MINUS; UNARY_PLUS; CASE_GROUP; ELIST; FOR_INIT; FOR_CONDITION;
    FOR_ITERATOR; EMPTY_STAT; FINAL="final"; ABSTRACT="abstract";
    STRICTFP="strictfp"; SUPERCTOR_CALL; CTOR_CALL;IF;
}

    private int ltCounter = 0;

compilationUnit
    : ( packageDefinition
       | */ nothing */
    )
    ( importDefinition )* ( typeDefinition )* EOF;

packageDefinition
    options {defaultErrorHandler = true;} // let ANTLR handle errors
    : p:"package" {#p.setType(PACKAGE_DEF);} identifier SEMI!
    ;

importDefinition
    options {defaultErrorHandler = true;}
    : i:"import" {#i.setType(IMPORT);} identifierStar SEMI!
    ;

typeDefinition
    options {defaultErrorHandler = true;}
    : m:modifiers!
       ( classDefinition[#m]
          | interfaceDefinition[#m]
       )
       | SEMI!
    ;
declaration!: m:modifiers t:typeSpec[false] v:variableDefinitions[#m,#t]
  {#declaration = #v;}
;

typeSpec[boolean addImagNode]
  : classTypeSpec[addImagNode]
  | builtInTypeSpec[addImagNode]
  ;

arraySpecOpt:
  (options{greedy=true;}: // match as many as possible
   lb:LBRACK ^ {# lb. setType(ARRAY_DECLARATOR);} RBRACK !
  )*;

classTypeSpec[boolean addImagNode]
  : classOrInterfaceType[addImagNode]
    arraySpecOpt
    {
      if ( addImagNode ) {
        #classTypeSpec = #(#[TYPE,"TYPE"], #classTypeSpec);
      }
    }
  ;

classOrInterfaceType[boolean addImagNode]
  : IDENT (typeArguments[addImagNode])?
    (options{greedy=true;}: // match as many as possible
     DOT !
    )* IDENT (typeArguments[addImagNode])?
  ;

typeArguments[boolean addImagNode]
  { int currentLtLevel = 0;}
  ;
    {currentLtLevel = ltCounter;}
  LT {ltCounter++;}
  classTypeSpec[addImagNode]
    (options{greedy=true;}: // match as many as possible
     ,COMMA classTypeSpec [addImagNode]
    )* ( options{generateAmbigWarnings=false;}: typeArgumentsEnd
     )?
  {((currentLtLevel != 0) || ltCounter == currentLtLevel)?
  ;

protected typeArgumentsEnd:
  GT {ltCounter-=1;}
  | SR {ltCounter-=2;}
  | BSR {ltCounter-=3;}
  ;

builtInTypeSpec[boolean addImagNode]
  : builtInType arraySpecOpt
if ( addImagNode ) {
    #builtInTypeSpec = #([TYPE,"TYPE"], #builtInTypeSpec);
}

type: classOrInterfaceType[false]
    | builtInType

builtInType:
    "void"
    | "boolean"
    | "byte"
    | "char"
    | "short"
    | "int"
    | "float"
    | "long"
    | "double"
    | "String"

identifier:
    IDENT ( DOT! IDENT )*

identifierStar:
    IDENT
    ( DOT! IDENT )*
    ( DOT! STAR )?

modifiers:
    ( modifier )*
    {#modifiers = #([MODIFIERS, "MODIFIERS"], #modifiers);}

modifier:
    "private"
    | "public"
    | "protected"
    | "static"
    | "transient"
    | "final"
    | "abstract"
    | "native"
    | "threadsafe"
    | "synchronized"
    | "volatile"
    | "strictfp"

classDefinition! [AST modifiers]
    : "class" IDENT
A.3 ANTLR Java 1.4 Grammar

```
(typeParameters)?
sc:superClassClause
ic:implementsClause
cb:classBlock
{#classDefinition = #([CLASS_DEF,"CLASS_DEF"],
    modifiers,IDENT,sc,ic,cb);}
;

superClassClause!: ( 
    "extends" classOrInterfaceType[false]
)?
;

interfaceDefinition!{AST modifiers]
    : "interface" IDENT
    (typeParameters)?
    ie:interfaceExtends
    cb:classBlock
{#interfaceDefinition = #([INTERFACE_DEF,"INTERFACE_DEF"],
    modifiers,IDENT,ie,cb);}
;
typeParameters
    { int currentLtLevel = 0; }
    : { currentLtLevel = ltCounter; }
    LT { ltCounter++; }
typeParameter (COMMA typeParameter)*
    (typeArgumentsEnd)?
    {{(currentLtLevel != 0) || ltCounter == currentLtLevel}? }
;
typeParameter
    : IDENT
    { options{generateAmbigWarnings=false;}: 
        "extends" classOrInterfaceType[false]
        (BAND classOrInterfaceType[false])* 
    }?
    ;
classBlock
    : LCURLY!
    ( field | SEMI! )*
    RCURLY!
{#classBlock = #([OBJBLOCK, "OBJBLOCK"], #classBlock);}
;
interfaceExtends
    : ( 
        e:"extends"!
        classOrInterfaceType[false] ( COMMA! classOrInterfaceType[false] )* 
    )?
{#interfaceExtends = #([EXTENDS_CLAUSE,"EXTENDS_CLAUSE"],
    #interfaceExtends);}
;
```
implmentsClause
: ( 
  i: "implements" ! classOrInterfaceType[false] ( COMMA ! classOrInterfaceType[false] ) * 
) ?

 #{implmentsClause = #(#[ IMPLEMENTS_CLAUSE," IMPLEMENTS_CLAUSE"],
        #implmentsClause);}

;

field!
: mods : modifiers

  ( h: ctorHead s: constructorBody // constructor
    {field = #(#[CTOR_DEF,"CTOR_DEF"], mods, h, s);}
  | cd: classDefinition[#mods] // inner class
    {field = #cd;}
  | id: interfaceDefinition[#mods] // inner interface
    {field = #id;}
  | (typeParameters)? t: typeSpec[false] // method or variable declaration(s)
    ( IDENT // the name of the method
    lp1 : LPAREN ^ argList RPAREN ! SEMI !
     {#lp1 . setType ( CTOR_CALL );#lp1 . setText ("CTOR_CALL ");}
    )
  | "static" s3: compoundStatement
    {field = #(#[STATIC_INIT,"STATIC_INIT"], s3);}
  | s4: compoundStatement
    {field = #(#[INSTANCE_INIT,"INSTANCE_INIT"], s4);}
  ;

constructorBody
: lc: LCURLY ^ {#lc. setType ( BLOCK );# lc. setText ("BLOCK");}

  ( options { greedy = true; } : explicitConstructorInvocation)?
  (statement)*
  RCURLY!

;

explicitConstructorInvocation
: "this" ! lp1 : LPAREN ^ argList RPAREN ! SEMI !

  {#lp1 . setNodeType (CTOR_CALL);#lp1 . setText("CTOR_CALL");}
  | "super" ! lp2 : LPAREN ^ argList RPAREN ! SEMI !

  {#lp2 . setNodeType (SUPERCTOR_CALL);#lp2 . setText("SUPERCTOR_CALL");}

;

variableDefinitions[AST mods , AST t]
: variableDeclarator[getASTFactory().dupTree(mods),
    getASTFactory().dupTree(t)]

  ( COMMA!
    variableDeclarator[getASTFactory().dupTree(mods),
        getASTFactory().dupTree(t)]
  )*
variableDeclarator![AST mods, AST t] 
  : id:IDENT d:declaratorBrackets[t] v:varInitializer
  {#variableDeclarator =
    #(#[VARIABLE_DEF,"VARIABLE_DEF"], mods, #(#[TYPE,"TYPE"],d), id, v);}
  ;

declaratorBrackets[AST typ]
  : {#declaratorBrackets=typ;}
    (lb:LBRACK ^ {# lb. setType (ARRAY_DECLARATOR);} RBRACK !)*
  ;

varInitializer
  : ( ASSIGN ^ initializer )?
  ;

arrayInitializer
  : lc:LCURLY ^ {# lc. setType (ARRAY_INIT);}
    ( initializer
      ( options {
        warnWhenFollowAmbig = false;
      }
      : COMMA ! initializer
    )* (COMMA !)?
    )? RCURLY !
  ;

initializer
  : expression
  | arrayInitializer
  ;

torHead
  : IDENT // the name of the method
    LPAREN ! parameterDeclarationList RPAREN !
    (throwsClause)?
  ;

throwsClause
  : "throws" ^ identifier ( COMMA ! identifier )* 
  ;

parameterDeclarationList
  : ( parameterDeclaration ( COMMA ! parameterDeclaration )* )?
  {#parameterDeclarationList =
    #(#[PARAMETERS,"PARAMETERS"],#parameterDeclarationList);}
  ;

parameterDeclaration!
  : pm:parameterModifier t:typeSpec[false] id:IDENT
    pd:declaratorBrackets[#t]
    {#parameterDeclaration =
      #(#[PARAMETER_DEF,"PARAMETER_DEF"],pm, #(#[TYPE,"TYPE"],pd), id);}
parameterModifier : (f: "final")? {
    #parameterModifier = #([MODIFIERS,"MODIFIERS"], f);
}

compoundStatement : lc: LCURLY {# lc. setType (BLOCK); lc. setText("BLOCK");} (statement)* RCURLY!

statement : compoundStatement
| (declaration) => declaration SEMI!
| expression SEMI!
| m:modifiers! classDefinition[#m]
| IDENT c: COLON ^ {# c. setType (LABELED_STAT);} statement
| "if" LPAREN! expression RPAREN! statement
| "for" LPAREN!
| forInit SEMI! // initializer
| forCond SEMI! // condition test
| forIter SEMI! // updater
| RPAREN! statement // statement to loop over
| "while" LPAREN! expression RPAREN! statement
| "do" statement "while"! LPAREN! expression RPAREN! SEMI!
| "break" (IDENT)? SEMI!
| "continue" (IDENT)? SEMI!
| "return" (expression)? SEMI!
| "switch" LPAREN! expression RPAREN! LCURLY!
( casesGroup )* RCURLY!
| tryBlock
| "throw" expression SEMI!
| "synchronized" LPAREN! expression RPAREN! compoundStatement
| "assert" expression (COLON! expression )? SEMI!
| s: SEMI {# s. setType(EMPTY_STAT);}
}

casesGroup : (options {greedy = true;}):
    aCase
    caseSList
    {# casesGroup = #([CASE_GROUP, "CASE_GROUP"], #casesGroup);}

A.3 ANTLR Java 1.4 Grammar

aCase
   : ("case" expression | "default") COLON !
   ;

400
caseSList
   : (statement)*
   {#caseSList = #([BLOCK,"BLOCK"], caseSList);}
   ;

405
forInit
   : ( (declaration)=> declaration | expressionList )?
   {#forInit = #([FOR_INIT,"FOR_INIT"], forInit);}
   ;

410
forCond
   : (expression)?
   {#forCond = #([FOR_CONDITION,"FOR_CONDITION"], forCond);}
   ;

415
forIter
   : (expressionList)?
   {#forIter = #([FOR_ITERATOR,"FOR_ITERATOR"], forIter);}
   ;

420
tryBlock
   : "try" compoundStatement
   (handler)*
   ( finallyClause )?
   ;

425
finallyClause
   : "finally" compoundStatement
   ;

430
handler
   : "catch" LPAREN ! parameterDeclaration RPAREN ! compoundStatement
   ;

435
expression
   : assignmentExpression
   {#expression = #([EXPR,"EXPR"], expression);}

440

expressionList
   : expression (COMMA ! expression)*
   {#expressionList = #([LIST,"ELIST"], expressionList);}

445

assignmentExpression
   : conditionalExpression
   ( ( ASSIGN |
      PLUS_ASSIGN |
      MINUS_ASSIGN |
      STAR_ASSIGN |
      DIV_ASSIGN
      )
   ;

450


Harmen Kastenberg 91
ANTLR Appendix A.

```
| MOD_ASSIGN |
| SR_ASSIGN |
| BSR_ASSIGN |
| SL_ASSIGN |
| BAND_ASSIGN |
| BXOR_ASSIGN |
| BOR_ASSIGN |

) assignmentExpression
)
?

conditionalExpression
: logicalOrExpression
   ( QUESTION assignmentExpression COLON! conditionalExpression )?
;

logicalOrExpression
: logicalAndExpression (LOR logicalAndExpression)*
;

logicalAndExpression
: inclusiveOrExpression (LAND inclusiveOrExpression)*
;

inclusiveOrExpression
: exclusiveOrExpression (BOR exclusiveOrExpression)*
;

exclusiveOrExpression
: andExpression (BXOR andExpression)*
;

andExpression
: equalityExpression (BAND equalityExpression)*
;

equalityExpression
: relationalExpression ((NOT_EQUAL | EQUAL ) relationalExpression)*
;

relationalExpression
: shiftExpression
   ( ( ( LT |
   | GT |
   | LE |
   | GE |
   ) shiftExpression
   )* 
   | "instanceof" typeSpec[true]
   )
;

shiftExpression
: additiveExpression ((SL | SR | BSR) additiveExpression)*
```
additiveExpression
  : multiplicativeExpression ((PLUS | MINUS) multiplicativeExpression)*
  ;

multiplicativeExpression
  : unaryExpression ((STAR | DIV | MOD) unaryExpression)*
  ;

unaryExpression
  : INC unaryExpression
  | DEC unaryExpression
  | MINUS {#MINUS . setType (UNARY_MINUS);} unaryExpression
  | PLUS {#PLUS . setType (UNARY_PLUS);} unaryExpression
  | unaryExpressionNotPlusMinus
  ;

unaryExpressionNotPlusMinus
  : BNOT unaryExpression
  | LNOT unaryExpression
  | (options {generateAmbigWarnings=false;}: lpb:LPAREN {#lpb . setType (TYPECAST);} builtInTypeSpec [true] RPAREN!
    unaryExpression
  | (LPAREN classTypeSpec [true] RPAREN unaryExpressionNotPlusMinus) =>
    lp:LPAREN {#lp . setType (TYPECAST);} classTypeSpec [true] RPAREN!
    unaryExpressionNotPlusMinus
    | postfixExpression
  )
  ;

postfixExpression
  :
    primaryExpression
    {
      DOT! IDENT
      ( lp:LPAREN {#lp . setType (METHOD_CALL);#lp . setText ("METHOD_CALL");}
        argList
        RPAREN!
      )?
      | DOT! "this"
      | DOT! "super"
      ( lp3:LPAREN argList RPAREN!
        {#lp3 . setType (SUPERCTOR_CALL);#lp3 . setText ("SUPERCTOR_CALL");}
      )
      | DOT! IDENT
      ( lps:LPAREN {#lps . setType (METHOD_CALL);#lps . setText ("METHOD_CALL");}
        argList
        RPAREN!
      )?
      | DOT! newExpression
      | lb:LBRACK {#lb . setType (INDEX_OP);} expression RBRACK!
    )
  | ( in:INC {#in . setType (POST_INC);}
ANTLR

Appendix A.

| de: DEC^ {# de.setType(POST_DEC);} |
| de: DEC^ {# de.setType(POST_DEC);} |

570

primaryExpression
: identPrimary ( options { greedy=true; } : DOT! "class" )?
| constant
| "true"
| "false"
| "null"
| newExpression
| "this"
| "super"
| LPAREN! assignmentExpression RPAREN!
| builtInType
( lbt:LBRACK^ {# lbt.setType(ARRAY_DECLARATOR);} RBRACK! )*
| DOT! "class"

585

identPrimary
: IDENT
(options { greedy=true; }:
  DOT! IDENT
)*
(options { greedy=true; }:
  ( lp:LPAREN^ {# lp.setType(METHOD_CALL);# lp.setText("METHOD_CALL");} argList RPAREN! )
  | ( options { greedy=true; }:
    lbc:LBRACK^ {# lbc.setType(ARRAY_DECLARATOR);} RBRACK! )
    )
)?

600

newExpression
: "new"^ type
( LPAREN! argList RPAREN! (classBlock)?
  | newArrayDeclarator (arrayInitializer)?
)

605

argList
: ( expressionList
  | {#argList = #[ELIST,"ELIST"];}
)

610

newArrayDeclarator
: (options { warnWhenFollowAmbig = false; }:
  lb:LBRACK^ {# lb.setType(ARRAY_DECLARATOR);}
  (expression)?
  RBRACK!
) *

615

constant
: NUM_INT
| CHAR_LITERAL
| STRING_LITERAL
Listing A.1: Antlr Java 1.4 grammar

| NUM_FLOAT |
| NUM_LONG |
| NUM_DOUBLE |
;
B.1 Installing GROOVE

To install GROOVE, download the ZIP-file of the latest GROOVE-version from http://www.cs.utwente.nl/groove. Unpack the ZIP-file to some convenient directory (this directory will further be referred to as the GROOVE-directory).

In the GROOVE-directory there are three batch-files named Editor.bat, Generator.bat and Simulator.bat. You need to change them at two places. Listing B.1 shows the content of the original batch-file Editor.bat. The other two files need to be adapted at exactly the same places.

```
@echo off
set APPL=Editor
set JDK="C:\Program Files\j2sdk1.4.1_02"
set LIB_DIR="C:\local\groove-0_2_1"
%JDK%\bin\java -jar %LIB_DIR%\%APPL%.jar %1 %2
```

Listing B.1: Editor.bat

You have to make sure that path on line 4 refers to the directory in which the java-executables can be found. Line 5 should refer to the GROOVE-directory.

B.2 Using GROOVE

By double-clicking the batch-files Editor.bat or Simulator.bat, the GROOVE-editor or -simulator is started. The editor can then be used to create graphs, which represent states
of the system, or rules that can be applied to transform graphs.

## B.3 Syntax

Creating graphs, and especially graph production rules, in GROOVE is done by adding nodes and edges to the graph created so far. When starting the GROOVE-editor you always start with the empty graph. By adding nodes and edges, the graph can be made more complex. After double-clicking on the elements of the graph (i.e. the nodes and edges), you can specify their labels. As mentioned before, when adding a label to a node, you actually add a self-edge to that node with the specified label.

The different visual representation of elements in graph production rules require a textual representation. Since we need a way to categorise nodes and edges as reader, eraser, embargo or creator, special role prefixes have been introduced. They are listed in Table B.1

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>(nothing)</td>
<td>Reader node or edge</td>
</tr>
<tr>
<td>use:</td>
<td>Reader node or edge</td>
</tr>
<tr>
<td>del:</td>
<td>Eraser node or edge</td>
</tr>
<tr>
<td>not:</td>
<td>Embargo node or edge</td>
</tr>
<tr>
<td>new:</td>
<td>Creator node or edge</td>
</tr>
</tbody>
</table>

Table B.1: Role prefixes in the textual representation for graph production rules.

Figure B.1 explains how these prefixes are used by showing the graphical and textual graph production rule put as already shown in Chapter 4.

(i) Graphical.  (ii) Textual.

Figure B.1: Graphical and textual representation of graph production rules.
B.3 Syntax

B.3.1 Regular Expressions

For specifying labels of edges using regular expressions, the following syntax must be used.

<table>
<thead>
<tr>
<th>Character</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>(nothing)</td>
<td>choice operator</td>
</tr>
<tr>
<td>.</td>
<td>sequence operator</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>chain operator (0..*)</td>
</tr>
<tr>
<td>+</td>
<td>chain operator (1..*)</td>
</tr>
</tbody>
</table>

Table B.2: Role prefixes in the textual representation for graph production rules.

Explaining the use of regular expressions is done by an example based on the concept of a simple list. The elements in the list can either be assigned the value 1, 2 or 3. New elements are always added at the end of the list. Elements of the list can be deleted from any place in the list.

Figure B.2 (i) shows one particular state of the list in which it has three elements, all pointing to a different value. Figures B.2 (ii), (iii) and (iv) show the graph production rules that can be applied on the list.

When taking a closer look on the rule named del-end.gpr we see that the edge between the List-node and the first element of the list is labelled \texttt{next*}. If you want to be sure that you always remove the last element of the list, you need a regular expression to match the element just before the last element (if such an element exists) and the last element that will be removed. For this matching we need regular expressions since it not known at forehand how many elements the list contains.
(i) List containing three elements.

(ii) append.gpr  (iii) del-mid.gpr  (iv) del-end.gpr

Figure B.2: Using regular expressions for specifying graph production rules.
Description Logic Grammar

C.1 description-logic.g

Listing on purpose not yet inserted (decreases the number of pages to be printed significantly).

```java
/**
 * Description Logic AL-UENC Recognizer
 *
 * Version 1.00 April 20, 2004 -- initial release
 */

class DesLogParser extends Parser {
    options {
        k = 2; // two token lookahead
        buildAST = true;
    }

tokens {
    R_OR; R_AND; R_COMPOSE; R_NOT; R_INV; R_TR_CLOS; R_REF_TR_CLOS; C_TOP; C_BOTTOM; C_OR; C_AND; C_NOT;
    C_ALL; C_SOME; C_AT_LEAST; C_AT_MOST; C_EXACTLY; AGGREGATE; A_GT; A_GE; A_LT; A_LE; A_EQUAL;
    PREDICATE; FEATURES; FEATURE;
    FEATUREFUNCTION; AGGFUNC; PARAMS;
}

eexpression
    : c1:"TOP" {
        #c1.setType(C_TOP);
    } |
    c2:"BOTTOM" {
        #c2.setType(C_BOTTOM);
    } |
    unionExpression
    ;

unionExpression
    : intersectExpression (c:"OR" { #c.setType(C_OR); } intersectExpression)*
    ;

intersectExpression
```
Description Logic Grammar

Appendix C.

: negationExpression (c:"AND" {#c.setNodeType(C_AND);} negationExpression)*
;

negationExpression 35 : (c:"NOT" {#c.setNodeType(C_NOT);})? valueResExpression
;

valueResExpression 40 : existQuantExpression
| c:"ALL" {#c.setNodeType(C_ALL);} relation DOT! existQuantExpression
;

existQuantExpression 45 : numberRestrExpression
| c:"SOME" {#c.setNodeType(C_SOME);} relation DOT! numberRestrExpression
;

numberRestrExpression 50 : predicateRestriction
| c1:"AT_LEAST" {#c1.setNodeType(C_AT_LEAST);} constant DOT! relation
| c2:"AT_MOST" {#c2.setNodeType(C_AT_MOST);} constant DOT! relation
| c3:"EXACTLY" {#c3.setNodeType(C_EXACTLY);} constant DOT! relation
;

predicateRestriction 55 : primaryExpression
| a1:predicate! LPAREN! a12:predicateArgList! RPAREN!
{#predicateRestriction = #(#[PREDICATE,"PREDICATE"],#(a1,a12));}
;

predicate 60 : p1:GT {#p1.setNodeType(A_GT);}
| p2:GE {#p2.setNodeType(A_GE);}
| p3:LT {#p3.setNodeType(A_LT);}
| p4:LE {#p4.setNodeType(A_LE);}
| p5:EQUAL {#p5.setNodeType(A_EQUAL);}
;

primaryExpression 70 : IDENT
| LPAREN! expression RPAREN!
;

relation 75 : intersectRelation (r:"REL_OR" {#r.setNodeType(R_OR);} intersectRelation)*
;

intersectRelation 80 : complementRelation (r:"REL_AND" {#r.setNodeType(R_AND);} complementRelation)*
;

complementRelation 85 : (r:"NOT" {#r.setNodeType(R_NOT);})? inverseRelation
;

inverseRelation 90 : (r:"INV" {#r.setNodeType(R_INV);})? composeRelation
composeRelation
: transClosRelation (r:"COMPOSE" {#r.setType(R_COMPOSE);} transClosRelation)*
;
transClosRelation
: (c:"TR_CLOS" {#c.setType(R_TR_CLOS);})? reflTransClosRelation
;
reflTransClosRelation
: (c:"REF_TR_CLOS" {#c.setType(R_REF_TR_CLOS);})? primaryRelation
;
primaryRelation
: f1:IDENT | LPAREN! relation RPAREN!
;
predicateArgList
: f1:featureChain! COMMA! f2:featureChain!
{#predicateArgList = #(#[FEATURES,"FEATURES"],#[FEATURE,"FEATURE"],f1),#([FEATURE,"FEATURE"],f2));}
;
featureChain
: feature (DOT! featureChain)? |
| f11:"countFirst"! LPAREN! f12:relation! RPAREN!
{#featureChain = #(#[AGGREGATE,"AGGREGATE"],f11,#(PARAMS,"PARAMS"),f12));}
| f21:"countSecond"! LPAREN! f22:relation! RPAREN!
{#featureChain = #(#[AGGREGATE,"AGGREGATE"],f21,#(PARAMS,"PARAMS"),f22));}
| f31:"mapToConstant"! LPAREN! f32:expression! COMMA! f33:constant! RPAREN!
{#featureChain = #(#[AGGREGATE,"AGGREGATE"],f31,#(PARAMS,"PARAMS"),f32,f33));}
| f41:"add"! LPAREN! f42:predicateArgList RPAREN!
{#featureChain = #(#[AGGREGATE,"AGGREGATE"],f41,#(PARAMS,"PARAMS"),f42));}
| f51:"sub"! LPAREN! f52:predicateArgList RPAREN!
{#featureChain = #(#[AGGREGATE,"AGGREGATE"],f51,#(PARAMS,"PARAMS"),f52));}
{#featureChain = #(#[AGGREGATE,"AGGREGATE"],f61,#(PARAMS,"PARAMS"),f62,f63));}
;
feature
: IDENT
;
aggFeatureName
: "min" |
| "max"
| "count"
| "sum"
| "average"
constant
  : NUM_INT
  ;

// The DesLog scanner
class DesLogLexer extends Lexer {
  options {
    testLiterals=false;
    k=4;
    charVocabulary='\u0003'..'\uFFFF';
    codeGenBitsetTestThreshold=20;
  }

// OPERATORS
LPAREN : '(';
RPAREN : ')';
COMMA : ',';
DOT : '.';
EQUAL : '==';
GE : '>=';
GT : '>'; 
LE : '<='; 
LT : '<';

IDENT
  options { testLiterals=true; }
  : ('a'..'z'|'A'..'Z'|'_'|'@'|'$') ('a'..'z'|'A'..'Z'|'_'|'0'..'9'|'@'|'$'|'.'|'\-')* 
  ;

NUM_INT
  : // decimal including zero 
    ('0'..'9') ('0'..'9')* 
  ;

Listing C.1: description-logic.g
Table D.1 defines the syntax as it needs to be used when passing DL-expressions to the evaluator.

<table>
<thead>
<tr>
<th>Conceptual constructors</th>
<th>Relational constructors</th>
</tr>
</thead>
<tbody>
<tr>
<td>⊤</td>
<td>“TOP”</td>
</tr>
<tr>
<td>⊥</td>
<td>“BOTTOM”</td>
</tr>
<tr>
<td>( C \sqcup D )</td>
<td>“OR” ( D )</td>
</tr>
<tr>
<td>( C \sqcap D )</td>
<td>“AND” ( D )</td>
</tr>
<tr>
<td>( \neg C )</td>
<td>“NOT” ( C )</td>
</tr>
<tr>
<td>( \forall R.C )</td>
<td>“ALL” ( R \ C )</td>
</tr>
<tr>
<td>( \exists R.C )</td>
<td>“SOME” ( R \ C )</td>
</tr>
<tr>
<td>≥ ( nR )</td>
<td>“AT_LEAST” ( n R )</td>
</tr>
<tr>
<td>≤ ( nR )</td>
<td>“AT_MOST” ( n R )</td>
</tr>
<tr>
<td>= ( nR )</td>
<td>“EXACTLY” ( n R )</td>
</tr>
</tbody>
</table>

Table D.1: Syntax to be used for constructing concepts and relations.
Graph Production Rules

E.1 Control Flow GPS

The rules that have been defined for performing the transformation process for adding control flow information to the abstract syntax tree fall apart in two categories. This has to do with the fact whether a statement being involved in the transformation rule is the last statement in a sequence of statements or not. If it is, the transformation rule needs to apply the \textit{sequence} and \textit{nesting} operation on the control flow graph as it is already created so far. If it is not, the transformation rule only has to enable the nesting-operation for the statement being involved. We will show all the rules that involve the transformation of statements being the last in a sequence. The two rules \texttt{expression.gpr} and \texttt{expression-sibling.gpr} show the difference between the transformation of statements according to their occurrence in a sequence (i.e. being the last statement in a sequence or not, respectively).
initial – step – method.gpr   initial – step – constructor.gpr

block.gpr   return.gpr

evaluation.gpr   evaluation – sibling.gpr
E.1 Control Flow GPS

if − then.gpr

for.gpr

while.gpr

variable − declaration.gpr

variable − instantiation.gpr
E.2 Call GPS

The rules involving the creation of the call structure in the class graph can be split up in three categories. The first category consists of rules triggering the rest of the rules. We need two such rules: one for methods and one for constructors. The other two categories distinguish between rules that create a new calling relation between procedures and rules that take care of situations in which the calling relation to be created already exists.

\[
\text{method} \rightarrow \text{def.gpr} \quad \text{constructor} \rightarrow \text{def.gpr}
\]

\[
\text{new} \rightarrow \text{ctor} \rightarrow \text{method} \rightarrow \text{call.gpr} \quad \text{exists} \rightarrow \text{ctor} \rightarrow \text{method} \rightarrow \text{call.gpr}
\]

\[
\text{new} \rightarrow \text{method} \rightarrow \text{ctor} \rightarrow \text{call.gpr} \quad \text{exists} \rightarrow \text{method} \rightarrow \text{ctor} \rightarrow \text{call.gpr}
\]
E.2 Call GPS

new - method - call.gpr  exists - method - call.gpr

new - ext - ctor - call.gpr  new - ext - method - call.gpr

Harmen Kastenberg
E.3 Dependence GPS

As described in Chapter 4, the graph grammar adding dependence information to the abstract syntax tree consists of two subgrammars. The first two rules are part of the first subgrammar; the third rule is the only rule in the second subgrammar.
Abstract Syntax Trees of DL-expressions

F.1 Fan-in and Fan-out

(i) Procedure’s fan-in. (ii) Procedure’s fan-out.
F.2 Cyclomatic Complexity

(iii) Procedure’s cyclomatic complexity.
(iv) Feature `classdef2One`.
(v) Feature classdef2NrOfProcs. (vi) Feature averageAccInstVars.
Results Prometrix

In order for Prometrix to generate the required output, we need to translate the Java-program into a equivalent Pascal-program. First, the source of the pseudo Pascal-program is listed, whereafter the output generated by Prometrix is shown.

G.1 calculator.pas

```pascal
program Calculator ( input , output );

type
  string = array[1..10] of char;

var
  a , b : integer ;
  result : integer ;
  oper : string ;

procedure Calculator (x , y: integer );
begin
  setA (x);
  setB (y)
end ;

procedure add ;
begin
  result := a + b
end ;

procedure mul ;
begin
  result := a * b
end ;

procedure sub ;
```
begin
  if (a > b) then
    result := a - b
  else
    result := b - a
end;

procedure setA(i: integer);
begin
  a := i
end;

procedure setB(i: integer);
begin
  b := i
end;

begin
  BufferedReader (InputStreamReader());
  oper := ";
  println ("enter oper");
  oper := readLine();
  while not equals (oper, "stop") do
    begin
      println ("first integer");
      x := parseInt (readLine());
      println ("second integer");
      y := parseInt (readLine());
      Calculator(x, y);
      if equals (oper, "add") then
        add()
      else if equals (oper, "sub") then
        sub()
      else if equals (oper, "mul") then
        mul()
      else
        println ("wrong oper");
        println (result);
        println ("enter oper");
        readLine ( oper )
    end
end.

Listing G.1: calculator.pas
G.2 Call graph information

Callgraph node information

<table>
<thead>
<tr>
<th>Module</th>
<th>File</th>
</tr>
</thead>
<tbody>
<tr>
<td>_main</td>
<td></td>
</tr>
<tr>
<td>21 nodes</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

Calls:

<table>
<thead>
<tr>
<th>Module</th>
<th>File</th>
</tr>
</thead>
<tbody>
<tr>
<td>bufferedreader</td>
<td></td>
</tr>
<tr>
<td>inputstreamreader</td>
<td></td>
</tr>
<tr>
<td>println</td>
<td></td>
</tr>
<tr>
<td>readline</td>
<td></td>
</tr>
<tr>
<td>equals</td>
<td></td>
</tr>
<tr>
<td>parseint</td>
<td></td>
</tr>
<tr>
<td>calculator</td>
<td>calculator.pas</td>
</tr>
<tr>
<td>add</td>
<td>calculator.pas</td>
</tr>
<tr>
<td>sub</td>
<td>calculator.pas</td>
</tr>
<tr>
<td>mul</td>
<td>calculator.pas</td>
</tr>
</tbody>
</table>

Callgraph node information

<table>
<thead>
<tr>
<th>Module</th>
<th>File</th>
</tr>
</thead>
<tbody>
<tr>
<td>calculator</td>
<td></td>
</tr>
<tr>
<td>3 nodes</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

Called by:

<table>
<thead>
<tr>
<th>Module</th>
<th>File</th>
</tr>
</thead>
<tbody>
<tr>
<td>_main</td>
<td>calculator.pas</td>
</tr>
</tbody>
</table>

Calls:

<table>
<thead>
<tr>
<th>Module</th>
<th>File</th>
</tr>
</thead>
<tbody>
<tr>
<td>setup</td>
<td>calculator.pas</td>
</tr>
<tr>
<td>setb</td>
<td>calculator.pas</td>
</tr>
</tbody>
</table>
## G.3 Flow graph information

### Flowgraph metrics for module `main`

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Size Metrics</strong></td>
<td></td>
</tr>
<tr>
<td>Number of nodes</td>
<td>21</td>
</tr>
<tr>
<td>Number of compressed nodes</td>
<td>12</td>
</tr>
<tr>
<td>Number of edges</td>
<td>24</td>
</tr>
<tr>
<td>Control flow length metric</td>
<td>20</td>
</tr>
<tr>
<td><strong>Local Structure Metrics</strong></td>
<td></td>
</tr>
<tr>
<td>I/O D-structured</td>
<td>1</td>
</tr>
<tr>
<td>Occurrences of D2 (IP-THEN)</td>
<td>0</td>
</tr>
<tr>
<td>Occurrences of D1 (IP-THEN-END)</td>
<td>3</td>
</tr>
<tr>
<td>Occurrences of D3 (WHILE LOOP)</td>
<td>1</td>
</tr>
<tr>
<td>Occurrences of D4 (WHILE EXIT LOOP)</td>
<td>0</td>
</tr>
<tr>
<td>Occurrences of explicit primes</td>
<td>0</td>
</tr>
<tr>
<td>Occurrences of case constructs</td>
<td>0</td>
</tr>
<tr>
<td>Biggest prime</td>
<td>4</td>
</tr>
<tr>
<td>Depth of nesting</td>
<td>4</td>
</tr>
<tr>
<td>External McCabe</td>
<td>1</td>
</tr>
<tr>
<td><strong>Overall Structure Metrics</strong></td>
<td></td>
</tr>
<tr>
<td>McCabe’s metric</td>
<td>5</td>
</tr>
<tr>
<td>Nuber’s metric</td>
<td>36</td>
</tr>
<tr>
<td>Lambda metric</td>
<td>20.00</td>
</tr>
<tr>
<td>Smallest Höchene WAPC</td>
<td>22.50</td>
</tr>
<tr>
<td>Product VMPA</td>
<td>29</td>
</tr>
<tr>
<td>Extended product VMPA</td>
<td>35</td>
</tr>
<tr>
<td><strong>Testability Metrics</strong></td>
<td></td>
</tr>
<tr>
<td>Statement testability</td>
<td>1</td>
</tr>
<tr>
<td>Branch testability</td>
<td>1</td>
</tr>
<tr>
<td>Simple paths</td>
<td>5</td>
</tr>
<tr>
<td>Visit-Reach-Loop paths</td>
<td>5</td>
</tr>
<tr>
<td>All paths</td>
<td>infinite</td>
</tr>
</tbody>
</table>

### Flowgraph metrics for module `sub`

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Size Metrics</strong></td>
<td></td>
</tr>
<tr>
<td>Number of nodes</td>
<td>4</td>
</tr>
<tr>
<td>Number of compressed nodes</td>
<td>4</td>
</tr>
<tr>
<td>Number of edges</td>
<td>4</td>
</tr>
<tr>
<td>Function length metric</td>
<td>2</td>
</tr>
<tr>
<td><strong>Local Structure Metrics</strong></td>
<td></td>
</tr>
<tr>
<td>I/O D-structured</td>
<td>1</td>
</tr>
<tr>
<td>Occurrences of D2 (IP-THEN)</td>
<td>0</td>
</tr>
<tr>
<td>Occurrences of D1 (IP-THEN-END)</td>
<td>1</td>
</tr>
<tr>
<td>Occurrences of D3 (WHILE LOOP)</td>
<td>0</td>
</tr>
<tr>
<td>Occurrences of D4 (WHILE EXIT LOOP)</td>
<td>0</td>
</tr>
<tr>
<td>Occurrences of explicit primes</td>
<td>0</td>
</tr>
<tr>
<td>Occurrences of case constructs</td>
<td>0</td>
</tr>
<tr>
<td>Biggest prime</td>
<td>4</td>
</tr>
<tr>
<td>Depth of nesting</td>
<td>1</td>
</tr>
<tr>
<td>Essential McCabe</td>
<td>1</td>
</tr>
<tr>
<td><strong>Overall Structure Metrics</strong></td>
<td></td>
</tr>
<tr>
<td>McCabe’s metric</td>
<td>2</td>
</tr>
<tr>
<td>Nuber’s metric</td>
<td>2</td>
</tr>
<tr>
<td>Lambda metric</td>
<td>2.00</td>
</tr>
<tr>
<td>Smallest Höchene WAPC</td>
<td>2.00</td>
</tr>
<tr>
<td>Product VMPA</td>
<td>3</td>
</tr>
<tr>
<td>Extended product VMPA</td>
<td>3</td>
</tr>
<tr>
<td><strong>Testability Metrics</strong></td>
<td></td>
</tr>
<tr>
<td>Statement testability</td>
<td>2</td>
</tr>
<tr>
<td>Branch testability</td>
<td>2</td>
</tr>
<tr>
<td>Simple paths</td>
<td>2</td>
</tr>
<tr>
<td>Visit-Reach-Loop paths</td>
<td>2</td>
</tr>
<tr>
<td>All paths</td>
<td>2</td>
</tr>
</tbody>
</table>
## G.4 Summary

### Summary table for application 'calculator'

<table>
<thead>
<tr>
<th>Metric</th>
<th>Maximum</th>
<th>Minimum</th>
<th>Mean</th>
<th>Std.Dev.</th>
<th>Median</th>
<th>Nulls</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of nodes</td>
<td>21</td>
<td>2</td>
<td>5.14</td>
<td>6.51</td>
<td>2.00</td>
<td>0</td>
</tr>
<tr>
<td>Compressed nodes</td>
<td>12</td>
<td>2</td>
<td>3.71</td>
<td>3.45</td>
<td>2.00</td>
<td>0</td>
</tr>
<tr>
<td>Number of edges</td>
<td>24</td>
<td>1</td>
<td>4.06</td>
<td>7.00</td>
<td>1.00</td>
<td>0</td>
</tr>
<tr>
<td>Length metric</td>
<td>20</td>
<td>1</td>
<td>4.14</td>
<td>6.51</td>
<td>1.00</td>
<td>0</td>
</tr>
<tr>
<td>Biggest prime</td>
<td>4</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>2.00</td>
<td>0</td>
</tr>
<tr>
<td>Depth of nesting</td>
<td>4</td>
<td>0</td>
<td>0.71</td>
<td>1.39</td>
<td>0.00</td>
<td>0</td>
</tr>
<tr>
<td>Baseline McCabe</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>McCabe's metric</td>
<td>5</td>
<td>1</td>
<td>1.71</td>
<td>1.39</td>
<td>1.00</td>
<td>0</td>
</tr>
<tr>
<td>Frather's metric</td>
<td>36</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>1.00</td>
<td>0</td>
</tr>
<tr>
<td>Lambda metric</td>
<td>20.00</td>
<td>1.00</td>
<td>-</td>
<td>-</td>
<td>1.00</td>
<td>0</td>
</tr>
<tr>
<td>Basili-Hutchens</td>
<td>22.04</td>
<td>1.00</td>
<td>-</td>
<td>-</td>
<td>1.00</td>
<td>0</td>
</tr>
<tr>
<td>Sum VNDAP</td>
<td>35</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>1.00</td>
<td>0</td>
</tr>
<tr>
<td>Product VNDAP</td>
<td>29</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>1.00</td>
<td>0</td>
</tr>
<tr>
<td>Ext. product VNDAP</td>
<td>35</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>1.00</td>
<td>0</td>
</tr>
<tr>
<td>State. testability</td>
<td>2</td>
<td>1</td>
<td>1.14</td>
<td>0.35</td>
<td>1.00</td>
<td>0</td>
</tr>
<tr>
<td>Branch testability</td>
<td>2</td>
<td>1</td>
<td>1.14</td>
<td>0.35</td>
<td>1.00</td>
<td>0</td>
</tr>
<tr>
<td>Simple paths</td>
<td>5</td>
<td>1</td>
<td>1.71</td>
<td>1.39</td>
<td>1.00</td>
<td>0</td>
</tr>
<tr>
<td>VSL paths</td>
<td>5</td>
<td>1</td>
<td>1.71</td>
<td>1.39</td>
<td>1.00</td>
<td>0</td>
</tr>
<tr>
<td>All paths</td>
<td>2</td>
<td>1</td>
<td>1.17</td>
<td>0.37</td>
<td>1.00</td>
<td>1</td>
</tr>
<tr>
<td>Fan-in</td>
<td>1</td>
<td>0</td>
<td>0.86</td>
<td>0.35</td>
<td>1.00</td>
<td>0</td>
</tr>
<tr>
<td>Fan-out</td>
<td>10</td>
<td>0</td>
<td>1.71</td>
<td>3.45</td>
<td>0.00</td>
<td>0</td>
</tr>
<tr>
<td>Fan-out ex. lib</td>
<td>4</td>
<td>0</td>
<td>0.86</td>
<td>1.46</td>
<td>0.00</td>
<td>0</td>
</tr>
</tbody>
</table>
CD-ROM

The CD-ROM accompanying this thesis contains a file named `index.html`. By double-clicking this file, you will be guided through the documentation that has been generated during this project.

You can, for example, find the GXL-files of the graph production rules specifying the graph transformations that have been performed and the documentation of the Java-classes that have been implemented for this project.