The TTC 2013 Flowgraphs Case

Dipl.-Inform. Tassilo Horn
horn@uni-koblenz.de

University Koblenz-Landau, Campus Koblenz
Institute for Software Technology
Universitätsstraße 1, 56070 Koblenz, Germany

February 15, 2013

Abstract

This case for the Transformation Tool Contest 2013 is about evaluating the scope and usability of transformation languages and tools for a set of four tasks requiring very different capabilities. Two tasks deal with typical model-to-model transformation problems, there's a model-to-text problem, there are two in-place transformation problems, and finally there's a task dealing with validation of models resulting from the transformations.

The tasks build upon each other, but the transformation case project also provides all intermediate models, thus making it possible to skip tasks that are not suited for a particular tool, or for parallelizing the work among members of participating teams.

All models and metamodels are provided in the EMF/Ecore format. However, user's of other modeling frameworks are also highly encouraged to participate. If requested, export tools can be written to serialize the models in other formats such as simple CSV files or GXL.

1 Objective of the Case

The objective of this case is to allow participants to demonstrate their transformation tool's flexibility, i.e., to show its usability for a very diverse set of tasks requiring different transformation capabilities. Although different capabilities are needed, all tasks are connected by their general topic: analysis and transformation in compiler construction.

All models and metamodels are provided in the EMF/Ecore format. However, user's of other modeling frameworks are also highly encouraged to participate. If requested, the case proponent is willing to help writing an export tool to other formats such as CSV or GXL [WKR02].

Task 1 deals with a typical model-to-model transformation problem. Given an abstract syntax graph of a Java program conforming to the very detailed EMFText[1] JaMoPP metamodel [HJSW09], a new model, the structure graph of the original program, conforming to a much more abstract and simple metamodel has to be generated. Embedded in this task is a model-to-text transformation. In the JaMoPP model, a simple statement like `int i = a + b;` is expressed as a whole bunch of interrelated objects. In the target model, we only want one single `SimpleStmt`, but its `txt` attribute should be set to the original Java syntax: `int i = a + b;`.

In task 2, the structure graph resulting from task 1 should be enhanced with control flow information, i.e., every statement or expression has to be connected with its possible control flow successors as defined by the Java semantics [GJS+12]. This is an in-place

[1]http://www.emftext.org/index.php/EMFText

transformation task which is suited for graph transformation tools but can also be tackled algorithmically.

Task 3 is similar to task 2 from a technical point of view, that is, it is also an in-place transformation. Based on the control flow graph resulting from task 2, data flow information has to be synthesized. This also requires some additions to the model-to-model transformation from task 1. Again, this task is suited to be tackled with graph transformations or algorithmically.

The context of task 4 is a bit offside the strict transformation context. Nevertheless, it tackles the important challenge of model validation. Given the fact that you as transformation engineer put all your efforts in the transformation tasks, can you offload testing to developers that have a good Java knowledge but know nothing about MDE, EMF, or whatever modeling technology you are using?

Because every task builds upon the results of previous tasks, the intermediate models are also provided to allow participants to defer or skip tasks not particulary suited for their tools, or to allow teams for developing solutions in parallel. For this reason, there is no distinction between *core* and *extension* tasks. To participate, only one arbitrary task has to be solved, but of course scoring high presumably requires solving more tasks.

2 Detailed Task Description

Structure of the Case Project. The case project is available on GitHub[2]. The top-level folder `ttc-2013-flowgraphs-case` contains several directories. The directory `desc` contains the case description your are reading right now.

The `metamodel` directory contains the target metamodel `FlowGraph.ecore` including PDF images with several views on it focusing on the specific tasks. `StructureGraph.pdf` shows the target metamodel parts relevant for task 1, `ControlFlowGraph.pdf` shows the metamodel classes important for task 2, and `DataFlowGraph.pdf` shows the metamodel classes important for task 3.

The `src` folder contains several Java classes. Every class contains just one single method. For every class, one model will be generated as source for task 1. If you need more models to test your transformations, simply put a new Java file in this directory.

The `jamopp` directory contains the JaMoPP-Parser as JAR file that generates EMF models conforming to the JaMoPP metamodel out of Java source code files.

Initially, the `models` directory is empty. Top-level, there are two shell scripts `gen-models-from-src.sh` (for Unices) and `gen-models-from-src.bat` (for Windows). When being run, they use JaMoPP to parse all Java files in `src` and create corresponding models (file extension `java.xmi`) in `models`. Those are valid source models for task 1. Additionally, the JaMoPP parser serializes the JaMoPP metamodel next to the model files. It is called `java.ecore`[3] and the source metamodel of task 1.

The `results` folder contains all intermediate and final target models including visualizations. You can use them to validate your own results, or use them as source models for later tasks in case you skip an earlier task.

Finally, the `evaluation` directory contains an OpenDocument spreadsheet that will be used for scoring the solutions during the evaluation.

2.1 Task 1: Structure Graph

The first task requires writing a model-to-model transformation. The source models are the Java abstract syntax graphs conforming to the JaMoPP metamodel that are created

[2]This case's project: `https://github.com/tsdh/ttc-2013-flowgraphs-case`

[3]It'll also create a file `layout.ecore` containing classes that can be used to annotate abstract syntax graphs with layout information such as indentation, but this is of no importance for this case.

by the `gen-models-from-src.{sh,bat}` scripts. The JaMoPP metamodel covers the complete syntax of Java 7. However, to restrict the size of the transformation, the elements actually contained in the source models is very limited. For every `*.java` file, the corresponding model contains one compilation unit containing exactly one class with exactly one method. The method may have parameters. In the method's body, there may be local variable declarations, arithmetic expressions (only `+`, `-`, `*`, and `/`), assignments, unary modification expressions (`i++;` and `i--;`), `return` statements, and blocks. There may be `if`-statements and `while`-loops with a boolean expression as condition. Statements may be labeled, and `break`/`continue` may be used with or without target label. Section A in the appendix lists all actually used concrete JaMoPP classes.

The target metamodel of the transformation is depicted in Figure 1.

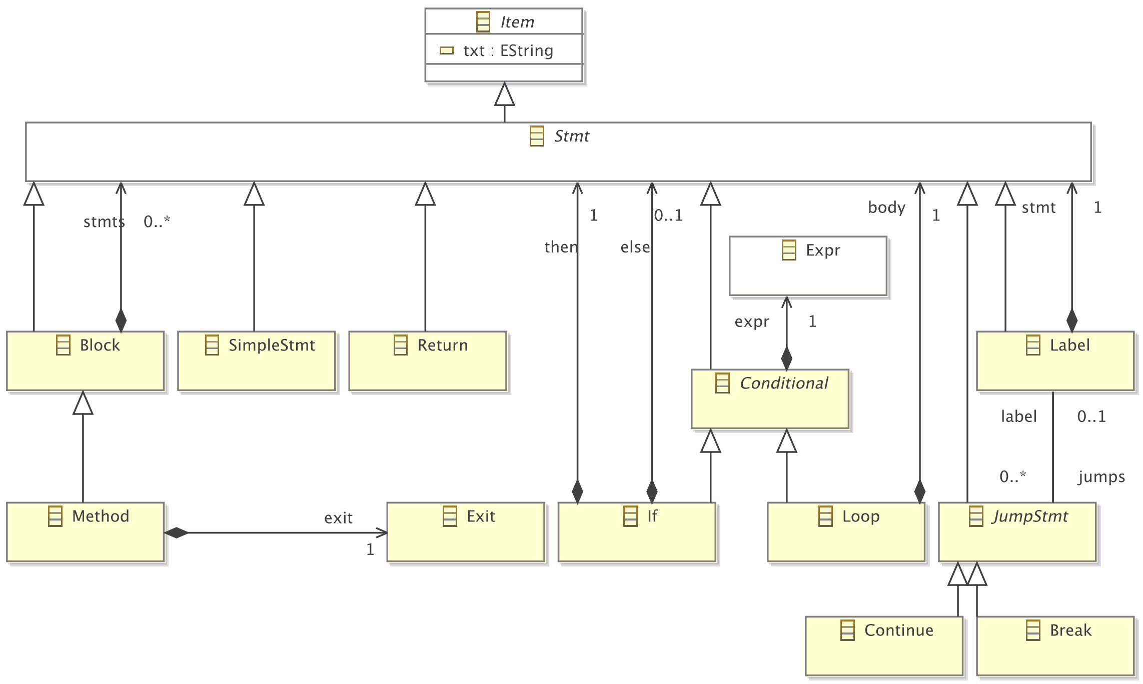

Figure 1: The target structure graph metamodel

The metamodel is very similar to the original JaMoPP metamodel from a structural point of view. The major difference is that statements and expressions are represented as one single object instead of being split up any further. Another difference is that every `Method` has exactly one `Exit`. There is no correspondence in Java, but it's a synthetic element added in favour of task 2. No matter how a method is exited, the last object in a method's control flow graph is this method's `Exit` object.

All metamodel classes extend the abstract `Item` class, even the class `Expr` although not visible in Figure 1 because an abstract, intermediate class between `Item` and `Expr` is not displayed for readabily reasons. `Item` declares a `txt` attribute. The transformation has to set the value of this attribute to the concrete Java syntax of the statement or expression, that is, there is a model-to-text transformation embedded in this model-to-model transformation. For example, for a local variable statement with a decimal integer literal as initial value like `int i = 17;` in the JaMoPP model, a `SimpleStmt` has to be created and its `txt` attribute has to be set to `"int i = 17;"`. Exceptions from this rule are all structured statements: the `txt` value of a `Block` is always `"{...}"`, the value for an `If` is always `"if"`, the value for a `Loop` is always `"while"`, the value for a `Method` with name `foo` is always `"foo()"` (no matter if there are parameters). The artificial `Exit` objects should have the value `"Exit"` set.

With the exception of `Break` and `Continue` objects that might refer to a target `Label`, the structure graphs created by the transformation are simple trees that reflect the containment hierarchy of the method.

For example, Listing 1 shows the method defined in `Test1.java`.

```java
public static void testMethod(int a) {
    int i = a * 2;
    i = i + 19;
    while (i > a) {
        if (a < 1) {
            return;
        } else if (a == 1) {
            break;
        }
        i--;
    }
}
```

Listing 1: An example Java method (`Test1.java`)

The resulting target model is visualized in Figure 4 in Appendix B on page 11.

2.2 Task 2: Control Flow Graph

This task deals with an in-place transformation problem. The semantics of the Java programming language should be integrated into the structure graphs created by the previous transformation. The task is to perform an intra-procedural control flow analysis. Any instruction should be connected to the instruction following it in the control flow.

Figure 2 shows the relevant metamodel excerpt.

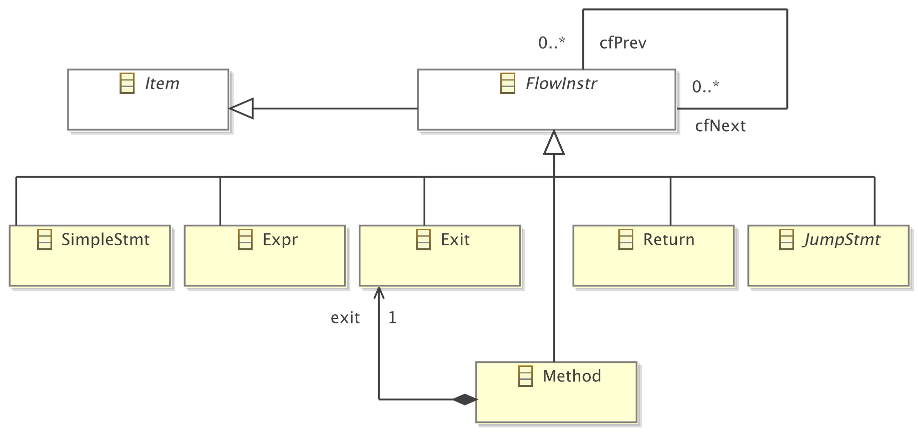

Figure 2: Metamodel classes related to control flow

Simple statements, expressions, the synthetical exits, methods, `return`, and the jump statements `break` and `continue` extend `FlowInstr`. Every flow instruction knows it immediate control flow predecessors (`cfPrev`) and successors (`cfNext`). It's those links the transformation has to synthesize from the structure graph.

Blocks, labels, loops, and if-statements don't participate in the control flow. Instead, when control flow reaches a block, the first flow instruction in the block is the control flow successor of the previous flow instruction. Since blocks may be nested in other blocks, the *first* flow instruction is actually the first one reachable by a depth-first search. These *first* semantics apply to whole description of this task.

In case of a label, the first flow instruction in the labled statement is the control flow successor.

In case of loops and if-statements, the successor is their test expression. This expression has in turn two control flow successors. If it is a test expression of a loop, the successors are the first flow instruction in the loop's body, and the first flow instruction following the loop. If it is a test expression of an if-statements, the first successor is the first flow instruction in then-statement. If there is an else-statement, its first flow instruction is the other control flow successor. Otherwise, the other successor is the first flow instruction in the statement following the if-statement.

The control flow successor of a `Method` is its first flow instruction, and `Return` statements always have the method's `Exit` as control flow successor.

The most complex control flow rules apply to the jump statements `Break` and `Continue`. Without a target label, the control flow successor of a `Break` is the first flow instruction following the immediately surrounding loop, and the successor of a `Continue` is the test expression of the immediately surrounding loop. With a target label `x`, the control flow successor of a `Break` is the first flow instruction following the statement labeled `x`, and the successor of a `Continue` is the expression of the surrounding loop labeled `x`[4].

The method in Listing 2 is the method defined in `Test2.java`.

```
public static void testMethod(int a) {
    int i = a * 2;
    while (i > a) {
        if (a < 1) {
            return;
        } else if (a == 0) {
            continue;
        }
        i++;
    }
}
```

Listing 2: An example Java method with complex control flow (`Test2.java`)

It's structure graph that is the result of task 1's transformation is the input to the control flow transformation. In case you haven't tackled task 1 yet, it is also included in this project as `results/Test2-StructureGraph.xmi`. The result control flow graph is visualized in Figure 5 in Appendix B on page 12. The control flow transformation should create a model equivalent to `results/Test2-ControlFlowGraph.xmi`.

2.3 Task 3: Data Flow Graph

In this task, an intra-procedural data flow analysis should be performed. This can be done based on the control flow graph, but currently one important piece of information is missing from it: for every flow instruction, we need to know which variables it reads and writes. Therefore, this task is twofold:

1. Extend the model-to-model transformation from task 1 so that it also creates `Var` objects for local variables and `Param` objects for method parameters, and connect each flow instruction to the variables it reads and writes.
2. Write a data flow transformation taking the control flow graph resulting from applying task 2's transformation on the result of the extended java-to-structure-graph transformation that synthesizes data flow links.

The relevant metamodel excerpt is shown in Figure 3.

[4]Interestingly, using `break x;` it is possible to jump out of any statement labeled `x`, i.e., in contrast to `continue`, the labeled statement doesn't need to be a loop.

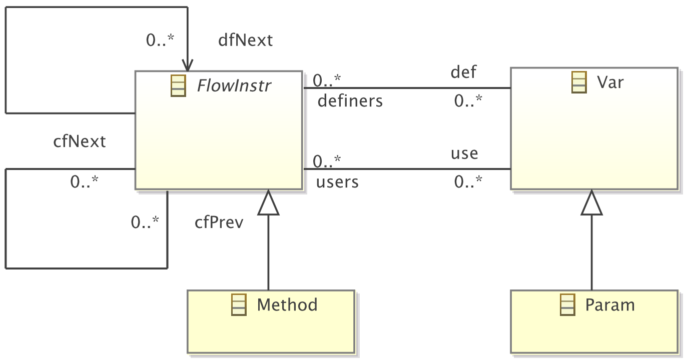

Figure 3: Metamodel classes related to data flow

Subtask 3.1. For every local variable statement in the JaMoPP model, the extended model-to-model transformation has to create a `Var` object. It's `txt` attribute should be set to the variable's name. Similarly, a `Param` object has to be created for every method parameter. Again, the `txt` attribute should be set to the parameter's name.

Furthermore, every flow instruction should be connected to the variables it writes (the `def` reference) and to the variables it reads (the `use` reference). The method parameters are the `def`-list of their method, the statement `a = b + c;` has a `def` link to `a`, and two `use` links to `b` and `c`, and the statement `i++;` both defines and uses `i`.

Thereafter, task 2's control flow transformation has to be applied to the result of the extended model-to-model transformation resulting in a structure graph where every flow instruction knows the variables/parameters it reads and writes, and which also contains the control flow links between flow instructions.

Again, the result models of this subtask are contained in the `results` folder. For every Java source file `TestX`, there is a model `TestX-ControlFlowGraph-with-Vars.xmi` plus a PDF visualization of it. In case you've skipped this subtask, you can use them to tackle the next subtask.

Subtask 3.2. The model resulting from applying the enhanced model-to-model transformation on the JaMoPP syntax graphs followed by applying the control flow transformation from task 2 to it is the source model for the data flow transformation to be developed in this subtask.

It's sole purpose is to synthesize `dfNext` links. For every flow instruction n, a `dfNext` link has to be created from all nearest control flow predecessors m that define a variable which is used by n. Formally:

$$\begin{aligned} m \rightarrow_{dfNext} n \iff & \; def(m) \cap use(n) \neq \emptyset \\ & \wedge \exists \, Path \; m = n_0 \rightarrow_{cfNext} \dots \rightarrow_{cfNext} n_k = n : \\ & (def(m) \cap use(n)) \setminus \left(\bigcup_{0<i<k} def(n_i) \right) \neq \emptyset \end{aligned}$$

That is, n uses at least one variable defined by m, and there is a control flow path from m to n in which at least one variable used by n and defined by m is not redefined by intermediate flow instructions.

There are several ways to tackle this problem. A simple one is to take the definition literally, i.e., for every flow instruction search the nearest control flow predecessors that define a variable used by instruction with quadratic worst-case effort. A more efficient and sophisticated algorithm is described in the dragonbook [ALSU06], chapter 9.1. The models resulting from this task which include control and data flow information are called *program dependence graphs* (PDG), and they play an important role in the optimization phase in compilers [FOW87].

The method defined in `Test0.java` is shown in Listing 3. The program dependence graph which this subtask's transformation has to generate is visualized in Figure 6 in Appendix B on page 13.

```java
public int testMethod() {
    int a = 1;
    int b = 2;
    int c = a + b;
    a = c;
    b = a;
    c = a / b;
    b = a - b;
    return b * c;
}
```

Listing 3: An example Java method for illustrating data flow (`Test0.java`)

The result models including visualizations are again available in the `results` directory. They are named `TestX-DataFlowGraph.{xmi,pdf}`. Because the `Var` objects were only needed for computing the data flow links, they are deleted from the models in order to keep the visualizations readable.

2.4 Task 4: Validation

The fourth task is no strict transformation task. Instead, the challenge is validating the control flow graphs created by task 2, and the program dependence graphs resulting from task 3. Concretely, it should be checked if all `cfNext` and `dfNext` links are set properly.

Since transformation experts have no time for testing their transformations in the same way as programmers have no time to test their programs, it would be fabulous if this boring work could be offloaded to other people. Assume those don't know anything about MDE in general or more specifically EMF (or whatever modeling framework you are using), but they are great Java programmers who can recite every section of the JLS [GJS+12] from the top of their heads. So the task is to give them a simple tool that gets a program dependence graph as input and all control and data flow links in an easy to write textual syntax. It should print all missing and all false links, i.e., all links defined in the textual specification that don't occur in the model, and all links occuring in the model that are not defined by the specification.

In the example Java programs provided in this case description project, every statement and expressions occurs at most once in a method, e.g., there's is no method with two `i++;` statements. As a result, for all PDGs generated from them, the `txt` attribute can be used to uniquely identify any object. This will be true for any additional methods that might be used for evaluation.

For example, such a specification for the method shown in Listing 3 could be written like in Listing 4. There's no restrictions on the actual syntax except that it should be easy to write for a Java programmer.

The requested tools's job is simple. It has to load a PDG and to read a specification such as depicted in Listing 4. For any `cfNext` or `dfNext` link in the model, it has to check if it is also defined in the specification. If not, it has to print a *false-link warning* message. Reversely, it has to check if every link defined in the specification also occurs in the model. If not, it has to print a *missing-link warning*.

```
cfNext: "testMethod()"   --> "int a = 1;"
cfNext: "int a = 1;"     --> "int b = 2;"
cfNext: "int b = 2;"     --> "int c = a + b;"
cfNext: "int c = a + b;" --> "a = c;"
cfNext: "a = c;"         --> "b = a;"
cfNext: "b = a;"         --> "c = a / b;"
cfNext: "c = a / b;"     --> "b = a - b;"
cfNext: "b = a - b;"     --> "return b * c;"
cfNext: "return b * c;"  --> "Exit"

dfNext: "int a = 1;"     --> "int c = a + b;"
dfNext: "int b = 2;"     --> "int c = a + b;"
dfNext: "int c = a + b;" --> "a = c;"
dfNext: "a = c;"         --> "b = a;"
dfNext: "a = c;"         --> "c = a / b;"
dfNext: "a = c;"         --> "b = a - b;"
dfNext: "b = a;"         --> "c = a / b;"
dfNext: "b = a;"         --> "b = a - b;"
dfNext: "c = a / b;"     --> "return b * c;"
dfNext: "b = a - b;"     --> "return b * c;"
```

Listing 4: An example textual DSL for validating the program dependence graph of the method shown in Listing 3

3 Evaluation Criteria

In the `evaluation` directory, there is an OpenDocument spreadsheet which will be used to score all submitted solutions (`evaluation_sheet.ods`).

There are two tables in the spreadsheet. The large upper one is for aggregating the individual scores of all voters to compute the overall ranking between all submitted solutions (*Overall Evaluation Sheet)*.

The small table below is for helping voters to assign a score in the range from 1 (not solved, or very poor) to 5 (excellent) for every task of each submitted solution (*Per-Task Evaluation Sheet)*. To score an individual task, voters should take into account the criteria reflected in that table. The most important one is *completeness & correctness*, so it is weighted with 50%. Another important factor weighted with 30% is *conciseness & understandability*. Lastly, the *efficiency* of the solution is another criterium that's weighted with the remaining 20%. To be able to evaluate efficiency, further much larger Java methods and corresponding models for test-driving the solutions will be provided. Of course, those will use only the exactly same Java language constructs as the existing examples in order not to break solutions that used to work fine with the already provided models.

For example, for a given solution's task 1, some voter assigns 5 points to completeness & correctness, 4 point to conciseness & understandability, and 3 points to efficiency. The table shows a total weighted score of 4.3 rounded to 4.

The voters score the tasks of all solutions in the same way and write down the weighted and rounded scores computed by the second table. Thus, every voter creates his personal score sheet that might look like Table 1.

These personal score sheets of all voters are then conveyed into the overall evaluation sheet. For every submitted solution and every task, the number of voters that assigned a score of $s \in [1, 5]$ is noted down in the cells of the table. The table then computes the average score per task for each solution and weights them to compute a total weighted score. Task 1 is weighted with 30%, Task 2 is weighted with 40% because its unquestionable the most complex one, and the tasks 3.1, 3.2, and 4 are weighted with 10% each. So

Voter i	Solution 1	Solution 2	...	Solution n
Task 1	4	3	...	3
Task 2	3	1	...	2
Task 3.1	4	4	...	1
Task 3.2	5	3	...	1
Task 4	2	5	...	2

Table 1: An example personal score sheet

the typical model-to-model transformation task 1 and its extension task 3.1 have a weight of 40% in total, the more graph transformation like tasks 2 and 3.2 have a total weight of 50%, and the validation task 4 with its 10% weight might be the one that tipps the scales.

References

[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319--349, July 1987.

[GJS$^+$12] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification*. California, USA, java se 7 edition edition, February 2012.

[HJSW09] Florian Heidenreich, Jendrik Johannes, Mirko Seifert, and Christian Wende. JaMoPP: The Java Model Parser and Printer. Technical Report TUD-FI09-10, Technische Universität Dresden, Fakultät Informatik, 2009. `ftp://ftp.inf.tu-dresden.de/pub/berichte/tud09-10.pdf`.

[WKR02] Andreas Winter, Bernt Kullbach, and Volker Riediger. An overview of the gxl graph exchange language. In *Revised Lectures on Software Visualization*, pages 324--336. Springer, 2002. `http://www.se.uni-oldenburg.de/documents/winter+2002.pdf`.

A List of used JaMoPP Classifiers

1. `classifiers.Class`
2. `containers.CompilationUnit`
3. `expressions.AdditiveExpression`
4. `expressions.AssignmentExpression`
5. `expressions.EqualityExpression`
6. `expressions.MultiplicativeExpression`
7. `expressions.RelationExpression`
8. `expressions.SuffixUnaryModificationExpression`
9. `literals.DecimalIntegerLiteral`
10. `members.ClassMethod`
11. `modifiers.Public`
12. `modifiers.Static`
13. `operators.Addition`
14. `operators.Assignment`
15. `operators.Division`
16. `operators.Equal`
17. `operators.GreaterThan`
18. `operators.LessThan`
19. `operators.MinusMinus`
20. `operators.Multiplication`
21. `operators.PlusPlus`
22. `operators.Subtraction`
23. `parameters.OrdinaryParameter`
24. `references.IdentifierReference`
25. `statements.Block`
26. `statements.Break`
27. `statements.Condition`
28. `statements.Continue`
29. `statements.ExpressionStatement`
30. `statements.JumpLabel`
31. `statements.LocalVariableStatement`
32. `statements.Return`
33. `statements.WhileLoop`
34. `types.ClassifierReference`
35. `types.Int`
36. `types.Void`
37. `variables.LocalVariable`

B Example Result Models

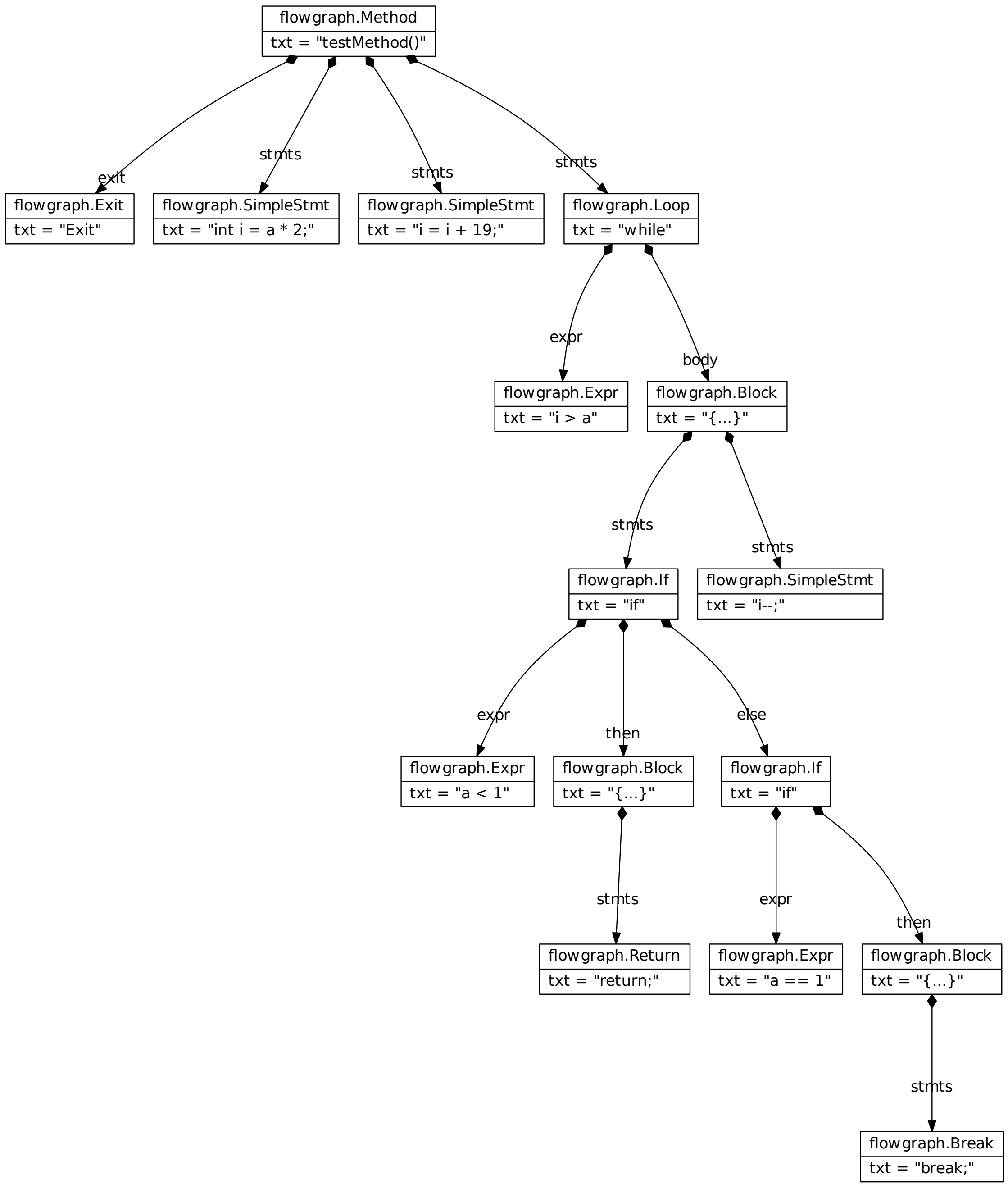

Figure 4: Structure graph corresponding to `Test1.java`

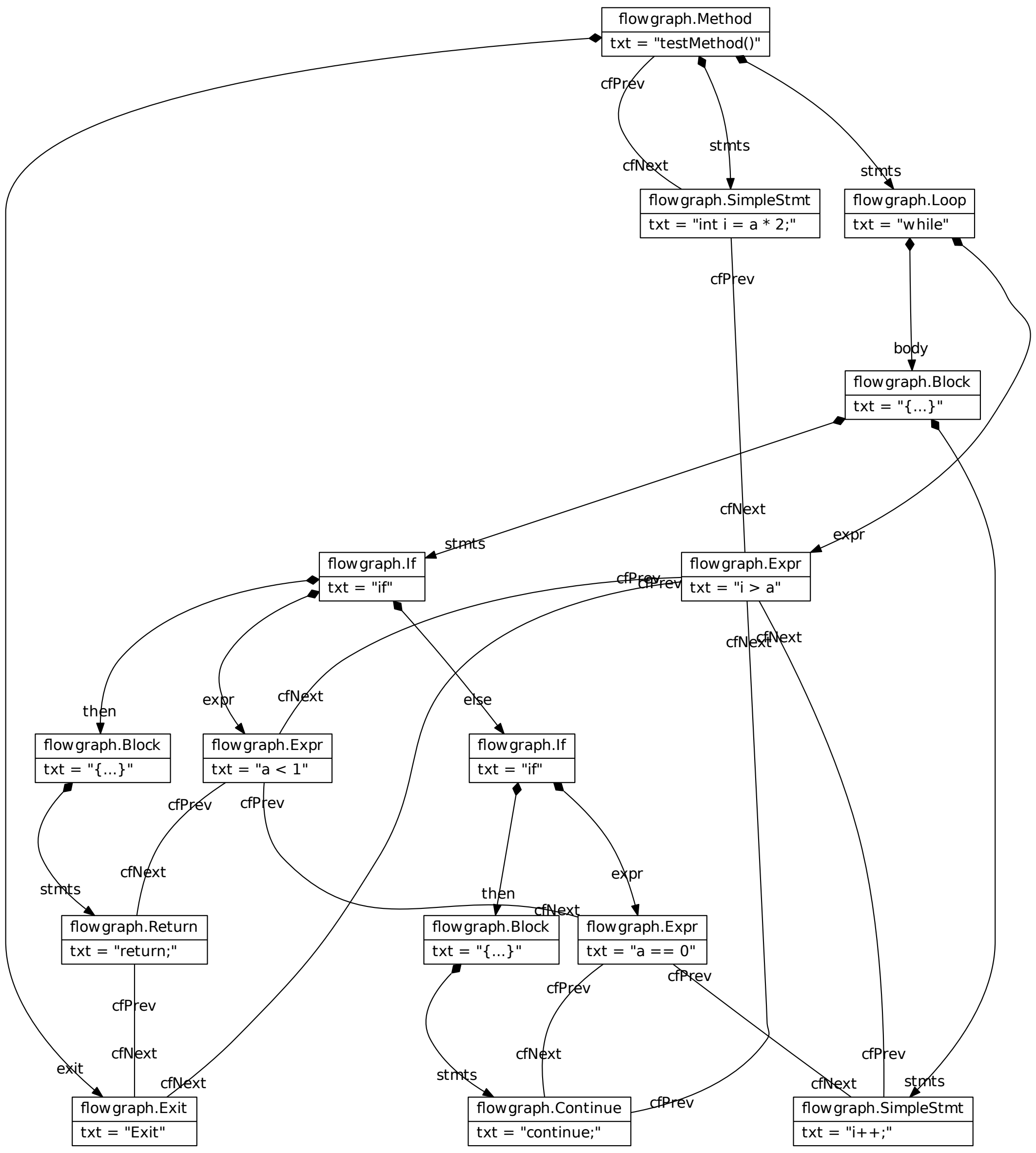

Figure 5: Control flow graph corresponding to `Test2.java`

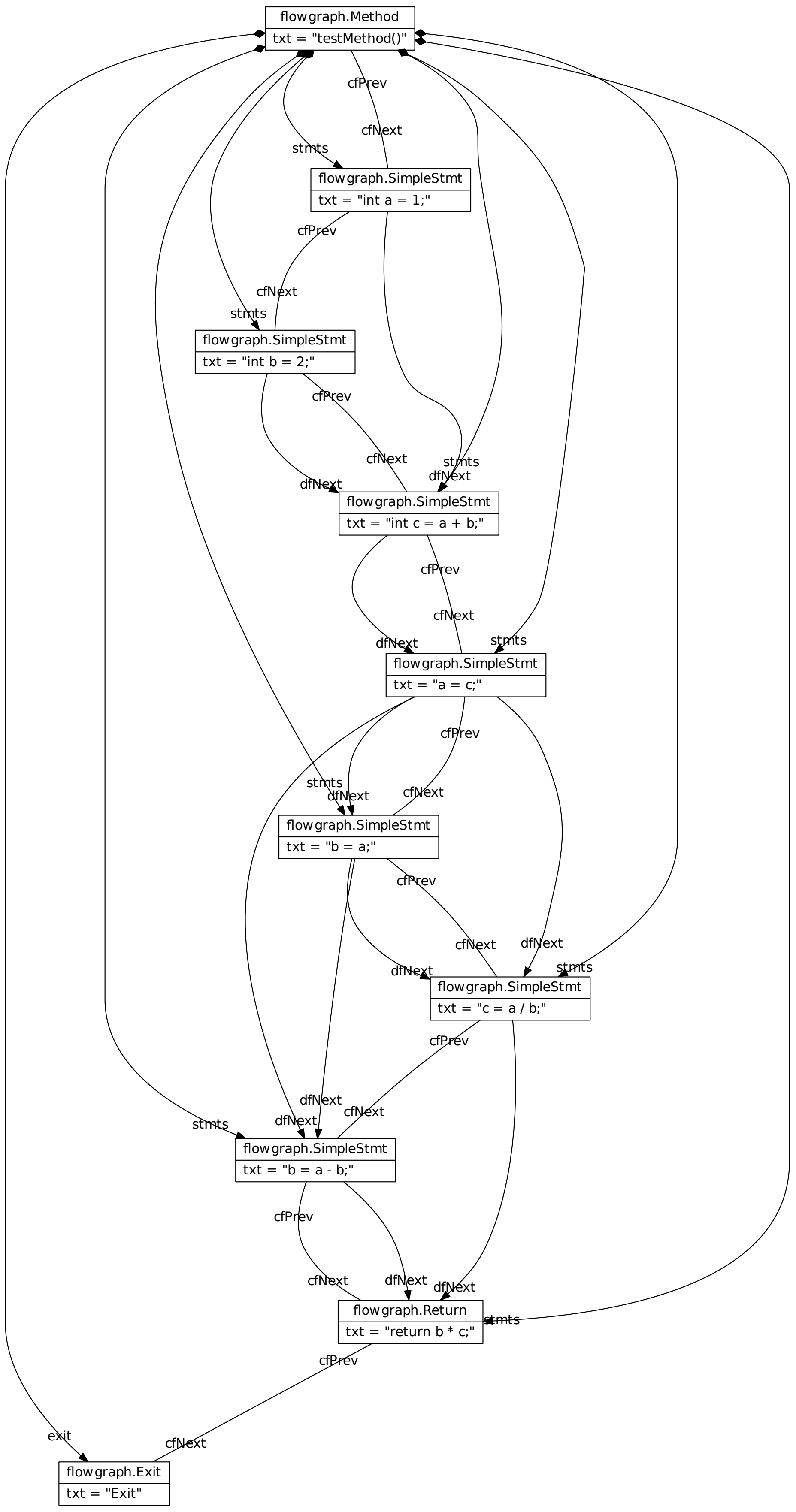

Figure 6: Program dependence graph corresponding to `Test0.java`