Program Comprehension

Jean-Sébastien Sottet and Frédéric Jouault

INRIA Centre Rennes Bretagne Atlantique,
Ecole des Mines de Nantes
4, rue Alfred Kastler, 44307, France
{jean-sebastien.sottet,frederic.jouault}@inria.fr
http://www.emn.fr/x-info/atlanmod/

Abstract. This paper presents a case study for the GraBaTs'09 tool contest. Program comprehension may benefit from graph and transformation techniques because they enable custom queries, and complex query results (i.e., graphs). However, there is one main issue to address: scalability. Software developers need tools that help them understand large code bases. For instance, the Java Development Tools from Eclipse consists of around six thousand Java classes, which translate into roughly five million nodes in the corresponding graph.
We propose two independent tasks in order to evaluate the adequacy of transformation tools to program comprehension and similarly demanding tasks. The first one exercises the scalability of tools by requiring them to filter large input graphs. The associated test-case contains a set of graphs of increasing size that should help comparing tools. The second task exercises the genericity of the tools. A tool that solves both tasks demonstrates that it scales, and that it does not trade scalability for genericity.

Key words: Program Comprehension, Program Query, Transformation Scalability

1 Introduction

This case study consists in the definition of program comprehension operators (i.e., queries over source code) using a graph or model transformation tool. Memory usage and execution time scalability with respect to input size, as well as a certain level of genericity[1] of these tools are exercised by two independent tasks.

In their daily activities, software developers heavily rely on preexisting code that has often been produced in different contexts, by different people. Most of the time, the associated documentation is either inexistent or out-of-date. Large source code bases cannot be comprehended by simply reading them sequentially. Therefore, software developers need tools that support them in understanding programs. However, typical development environments only provide limited and

[1] The genericity we consider here consists of the ability to: 1) use different source and target metamodels, and 2) chain several transformations.

predefined query support. Moreover, solutions that enable custom queries like JTL [1], Codequest [2], and JQuery [3] are limited to a single programming language (e.g., Java for the three mentioned tools). Because they are generally metamodel-independent, graph and model transformation techniques enable the customization of queries over source code for any language.

One of the main challenges in the definition of program comprehension operators as transformations is scalability with respect to input size. The two following examples give some concrete figures:

- **The Java standard library** source code that comes with version 1.6.0.04 of the Java Development Kit contains around nine thousand and five hundred (9500) classes. A graph representing its decorated Abstract Syntax Tree contains almost six million nodes.
- **The Eclipse Java Development Kit compiler** consists of around six thousand classes, and the corresponding graph requires almost five million nodes.

At the time of writing this paper, a typical development workstation is a dual-core 2.5 GHz 32 bits processor with 2GB of memory. On such a machine we need to run queries over large volumes of code (frequently more than 1000 classes) with rapid reporting. The result of one query may itself trigger the need to formulate another one and rapidly get the new result. The two main scalability problems are therefore:

- **Memory** (or ability to handle large graphs). Although some queries may be independently executed on different parts of a graph, some queries span over an entire code base. For this reason it becomes necessary to be able to process large graphs.
- **Time** (or Execution speed). Because of the generally interactive nature of program comprehension queries, rapid execution over large graphs is also required.

Other challenges reside in parsing source code into graphs or models, as well as in visualizing query results. However, these are beyond the scope of this case study. Already parsed Java programs are made available as models, and the results may be delivered as abstract graphs (visualization being optional).

This paper is organized as follows. Section 2 presents an overview of the two tasks that exercise scalability, and genericity. These tasks are further detailed in Sections 3, and 4. Finally, Section 5 gives the concluding remarks.

2 Overview of Tasks

The case study is divided into two tasks that may be solved independently:

- **Task 1: a simple filtering query** that selects a subgraph of its input according to a given condition. This task exercises the scalability of the tool on large graphs.

- **Task 2: a complex query results** that computes a Control Flow Graph (CFG) and a Program Dependence Graph (PDG). The purpose of this task is to demonstrate that the tool is able to return complex query results, and further operate on these results. Only small graphs are considered.

A tool that is only able to solve Task 1 is scalable but not generic. Existing tools like JTL, and JQuery have shown that scalability is possible when there is no requirement on genericity (i.e., they have a fixed source language, and query results cannot be complex graphs but only lists). Solving only Task 2 does not require scalability. A solution that provides answers to both tasks is therefore especially interesting. A grading scheme such as the following may be used in order to stress this point: one point for solving either Task 1 or Task 2, and five points for solving both.

Conceptually, Tasks 1 and 2 could be chained as illustrated on Figure 1: a first step selects some elements that are further processed in the second step. However, in order to decouple both tasks, they will in practice be solved separately in the context of this case study. To this end, the test-cases contain a set of large base models for Task 1, as well as a separate smaller model for Task 2.

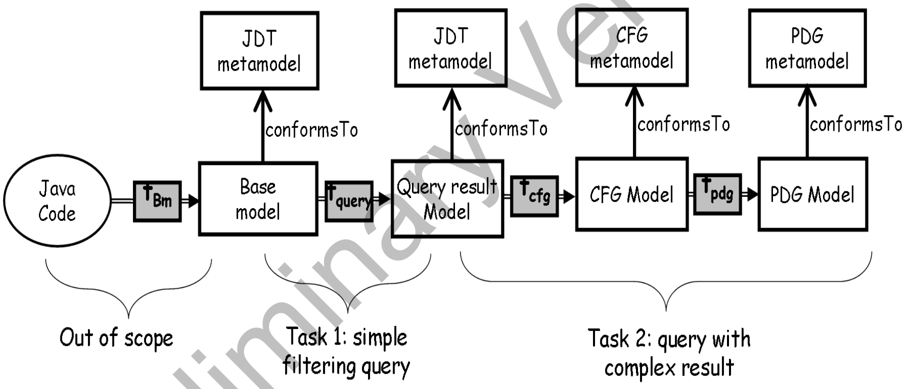

Fig. 1. Benchmark Tasks Overview

The graphs that will be queried represent decorated (i.e., with type information) abstract syntax trees of Java programs. The test-cases are provided[2] as models serialized in XML Metadata Interchange (XMI) 2.0 that is notably compatible with the Eclipse Modeling Framework (EMF) parser (although EMF

[2] Test-cases (metamodels and models) are available at: `http://www.emn.fr/x-info/atlanmod/index.php/GraBaTs_2009_Case_Study`. Each file (metamodel or model) may be downloaded separately, the larger models being available in both uncompressed, and compressed (zip & bzip2) forms. Additionally, an all-in-one archive may be downloaded (available in zip & bzip2).

may not be able to load all of them on just any machine, notably depending on available memory). Task 1 will operate on a set of graphs of increasing size, which should help comparing tools with each other (i.e., some will only work with the first set, and others will also work with larger sets). The metamodel to which the graphs conform is provided with the test-cases in two formats: in KM3 [4] with file *JDTAST.km3*, and in Ecore (i.e., the metametamodel from EMF) with file *JDTAST.ecore* (serialized in EMF XMI 2.0). It is an updated version of the metamodel used in the *SharenGo Java Legacy Reverse-Engineering* MoDisco use case[3].

This metamodel is an almost direct translation of the Eclipse Java Development Tools (JDT)[4] Application Programming Interface (API). More precisely, it corresponds to packages *org.eclipse.jdt.core* and *org.eclipse.jdt.core.dom*. The JavaDoc API documentation of these packages[5] may be used to better understand the metamodel.

3 Task One: Simple Filtering Query

This is a typical task in program comprehension: the developer queries the model to answer questions about the code, and starts by focusing on a region of interest that is first selected.

The tool **must** be able to select the set of classes that declare public static methods whose return type is themselves. It must only keep a class C if C declares a public static method whose return type is C. This query is presented in [1] where it is written as: `q1:= eq[T] declares:{ public static T(*); }`. The tool shall use the different test-cases of increasing size that are provided. A given tool may only work correctly with smaller sets, and fail with larger sets. Solution evaluation will take into account this information.

The main evaluation criteria of Task 1 is scalability. Several parameters are to be taken into account to evaluate a solution. These include the number of model elements that can be transformed, but also the memory required to do so. For instance, for a solution implemented in Java the following parameters will notably be considered: heap size and/or addressing space, in addition to execution time. Switching from Windows to Linux, or from a 32 bits Operating System (OS) to a 64 bits OS may be used to achieve the transformation[6]. However, all other parameters being equal, a solution that works under more restrictive constraints will obtain a better grade. Solutions should be presented

[3] The MoDisco use case is available from `http://www.eclipse.org/gmt/modisco/useCases/JavaLegacyRE/`.

[4] More information about JDT is available at: `http://www.eclipse.org/jdt/core/index.php`.

[5] The JDT JavaDoc is available from: `http://help.eclipse.org/stable/index.jsp?topic=/org.eclipse.jdt.doc.isv/reference/api/overview-summary.html`.

[6] For instance, the addressing space available for the Sun HotSpot Java virtual machine running on a 32 bits version of Windows is about 1.5GB (see `http://java.sun.com/docs/hotspot/HotSpotFAQ.html#gc_heap_32bit`)

with all relevant information (e.g., operating system type and word size, physical memory, used addressing space).

Other criteria like usability will also be considered: how easy is it to write the query?

4 Task Two: Complex Query Results

A tool that solves Task 2 demonstrates that it can work with different source and target metamodels, as well as further transform the result of a first query.

The tool **must** compute the CFG that represents the control flow dependencies between code elements: the order of execution of the program elements. Table 1 gives an example of CFG (on the right-hand side) for a small piece of code (on the left-hand side). This example is extracted from [5]. A CFG is a directed graph in which nodes correspond to statements, and an edge goes from each node to its immediate follower in execution order. The node corresponding to a conditional statement (e.g., *if*, *while*) has two outgoing edges labeled *true* and *false* that point to the immediate followers when the condition is respectively either *true* or *false*. The CFG metamodel is provided as part of the test-cases (i.e., files *CFG.km3*, and *CFG.ecore*).

```
prod = 1;
k= 1;

while(k<=10){
  prod = prod *k;
  k++;
}

System.out.print(k);
System.out.print(prod);
```

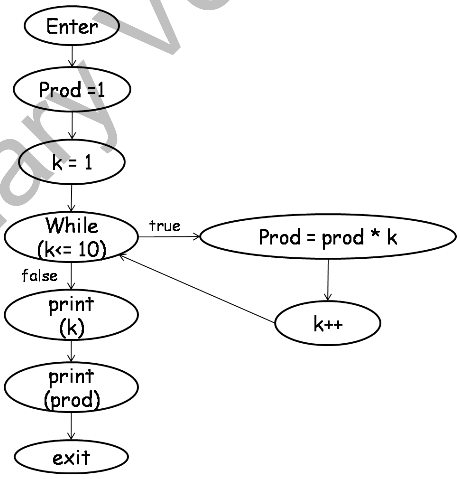

Table 1. Example of Control Flow Graph of a Simple Program

Furthermore, the tool **must** compute the Program Dependence Graph [6] from the CFG. The PDG represents both control flow dependency similarly to the CFG, but also data dependencies. Control flow of a PDG is a bit different from a CFG: it has a start node that decomposes the control dependencies as a tree (see Figure 2). For example the affectation of *k=1* affects all node that contains occurrences of k that are "after" in the CFG graph. Note that the *prod*

*= prod*k* do not affect other nodes that contains k (it do not modify k). The data dependence links represent all the influences between parameter, identifiers, etc. The PDG metamodel is provided as part of the test-cases (i.e., files *PDG.km3*, and *PDG.ecore*).

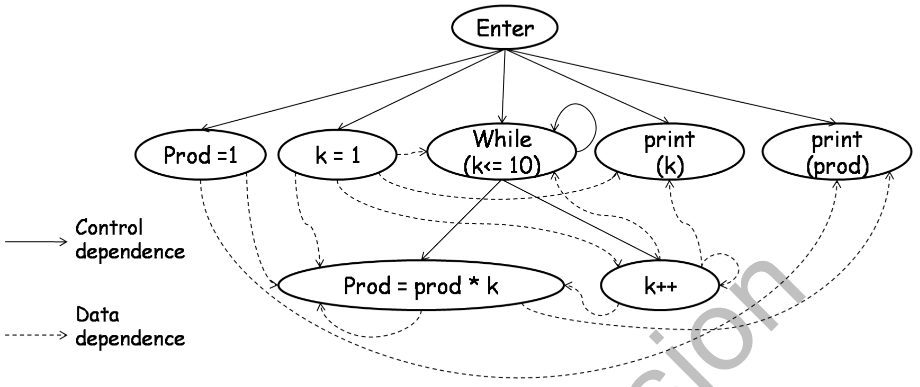

Fig. 2. Program Dependence Graph

Optionally, the tool **may** be able to graphically represent query results. If it does so, then it shall be able to output visualizations that are similar to the right-hand side of Table 1, and to Figure 2.

The main evaluation criteria of Task 2 is the ability to implement the transformations. As for Task 1, usability will also be considered: how easy is it to write these transformations?

5 Conclusion

Two complementary tasks that exercise different aspects of a transformation tool have been presented.

The first task consists of a relatively simple transformation over large input graphs, and exercises the scalability of transformation tools. Solving this task for small graphs should not be difficult. However, being able to handle large graphs such as those provided in the test-cases is not a simple matter.

For the second task, only simple graphs are considered, but the transformation is more complex. Moreover, a tool that solves this task has to be able to handle different source and target metamodels, as well as to be able to chain transformations.

Depending on the implementation of transformation tools, either task or both should be challenging. A tool that solves both demonstrates scalability that is not attained at the expense of genericity.

Acknowledgments. The authors would like to thank all the AtlanMod team, and more particularly Guillaume Doux, Mickaël Barbero and Jean-Bézivin. This work has been partially supported by the Lambda project.

References

1. T. Cohen, J. Y. Gil, and I. Maman. Jtl: the java tools language. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 89–108, New York, NY, USA, 2006. ACM.
2. E. Hajiyev, M. Verbaere, and O. de Moor. Codequest: Scalable source code queries with datalog. *Proc. of the 20th European Conference on Object-Oriented Programming*, 4067:2–27, 2006.
3. D. Janzen and K. De Volder. Navigating and querying code without getting lost. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 178–187, New York, NY, USA, 2003. ACM.
4. F. Jouault and J. Bezivin. Km3: a dsl for metamodel specification. In *Proceedings of 8th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems, LNCS 4037*, 2005.
5. S. Kumar and S. Horwitz. Better slicing of programs with jumps and switches. In *In Proc. of FASE 2002: Fundamental Approaches to Softw. Eng.*, 2002.
6. K. Ottenstein and L. Ottenstein. The program dependence graph in software development environement. In *Proc. of ACM SIGSOFT/SIGPLAN Software Engineering Symp. on Pratical Software Development Environments*, pages 177–184, 1984.