Injecting Execution Traces into a Model-Driven Framework for Program Analysis

Thibault Béziers la Fosse(✉), Massimo Tisi, and Jean-Marie Mottu

AtlanMod Team (Inria, IMT Atlantique, LS2N), Nantes, France
{thibault.beziers-la-fosse,massimo.tisi,jean-marie.mottu}@inria.fr

Abstract. Model-Driven Engineering (MDE) has been successfully used in static program analysis. Frameworks like MoDisco inject the program structure into a model, available for further processing by query and transformation tools, e.g., for program understanding, reverse-engineering, modernization. In this paper we present our first steps towards extending MoDisco with capabilities for dynamic program analysis.

We build an injector for program execution traces, one of the basic blocks of dynamic analysis. Our injector automatically instruments the code, executes it and captures a model of the execution behavior of the program, coupled with the model of the program structure. We use the trace injection mechanism for model-driven impact analysis on test sets. We identify some scalability issues that remain to be solved, providing a case study for future efforts in improving performance of model-management tools.

Keywords: Model-Driven Engineering · Dynamic program analysis
Execution traces

1 Introduction

Many properties of a program have to be analyzed during its lifecycle, such as correctness, robustness or safety. Those behaviors can be analyzed either dynamically by executing the program, or statically, usually by examining the source code. Several program analysis frameworks exist and are heavily used by the engineering community. Code coverage frameworks are a very popular example for dynamic analysis since they can significantly improve testing quality. Dynamic information can be used to add more precision to static analysis [5,6].

The MoDisco framework [4] is designed to enable program analysis in Model-Driven Engineering (MDE) by creating a model of the source code, and using it for program understanding and modernization. The code model makes the program structure easily accessible to external modeling tools for any kind of processing, e.g. for static analysis. This uniform treatment of models and programs in the MDE technical space has proven to greatly help the understanding, development and maintenance of complex software [3].

MDE tools analyzing source code through MoDisco do not usually execute the original program. However, if MoDisco provided models of dynamic aspects of the code, this would enable other useful analysis. For instance, a model of test traces for a *software under test* (SUT) and its test suite could foster a better understanding of the test suite, and of the impact that a statement modification could have on the test cases executing it. Moreover, it could be used to get an overview of the testing code coverage. Indeed, if the modification of a statement has no impact on any test method, the statement is considered as uncovered. Naturally this analysis model could provide a significant improvement to the quality of the test sets [11].

In this paper we show a method to build a model of the program execution, using both static and dynamic analysis. Thus, an initial structural model is statically built, containing the basic blocks of a program: packages, classes, methods and statements. Thereafter, the code is instrumented in order to add execution traces to the model during program execution. Consequently, each test method is associated to the ordered sequence of statements it executes.

The remainder of this paper is organized as follows. Section 2 presents our approach for model creation. Section 3 proposes an evaluation of the process on a set of projects having different sizes. Related work is discussed in Sect. 4, and Sect. 5 concludes this paper.

2 Approach

This section illustrates our approach for dynamically building a model of the program execution. We propose an automatic process made of a sequence of four steps. Figure 1 gives an overview of this process. On the left-hand side of Fig. 1, the input is the source code of the considered system (e.g. a program under test and its tests). First, a static model is generated thanks to a reverse engineering step (Sect. 2.1). Second, on the right-hand side of Fig. 1, the static model is refactored into an analysis model by a model transformation (Sect. 2.2). Third, on the left-hand side of Fig. 1, a source code instrumentation step (Sect. 2.3) prepares the code before execution. Finally, the instrumented code is executed and its instrumentation allows us to complete the analysis model into the dynamic model of the source code (Sect. 2.4). This model is the output of the process.

The dynamic model should contain the structure of the source code, especially describing the targeted system (e.g. the system under test). Furthermore it should reify which statements are executed when the system is run under the action of a launcher (e.g. a set of tests). In addition, the order of the calls should be stored to be used when analyzing the behavior of the system based on the dynamic model.

In this work, we consider Java source code and we exemplify the approach by the two classes in Fig. 2. The considered system is the class under test *Factorial* (left of Fig. 2) associated to a JUnit test class *TestFactorial* (right of Fig. 2). Those classes are the source code entry of the process in Fig. 1.

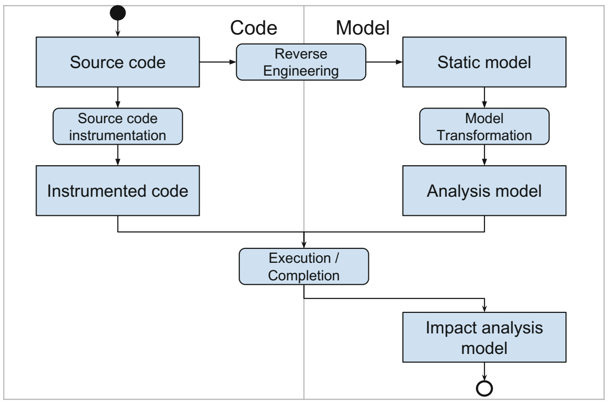

Fig. 1. 4-step process generating an test impact analysis model from source code

```
package main;

public class Factorial {

   public int fact(int n) {
      if (n == 1) {
         return 1;
      }
      return n * fact(n - 1);
   }
}
```

```
package main;

import static org.junit.Assert.*;
import org.junit.Test;

public class TestFactorial {
   @Test
   public void checkFact() {
      Factorial f= new Factorial();
      assertEquals(1, f.fact(1));
   }
}
```

Fig. 2. Factorial class, and its incomplete test suite.

2.1 Model Driven Reverse Engineering

The first step of our approach consists of generating the model of the code structure using MoDisco[1]. MoDisco has a visitor-based system which navigates through the full abstract syntax tree (AST), and then builds a model from it, according to its Java meta-model [4]. We use a specific option of MoDisco, which annotates each element of the output model with its location in the source code. This information will be necessary for the dynamic analysis, as we show later.

Figure 3 shows a simplified version of the model generated by MoDisco from the code in Fig. 2 (*Static model* in Fig. 1). *Node* elements contain the position, and a reference to the statements. Since the full static model generated by MoDisco is rather large, we extract the excerpt in Fig. 3, focusing on statement-level information, in order to improve performance. For instance we filter out information about expressions, binary files, or import declarations. Since in this work we focus on the execution trace of the statements, only those ones are needed, within their respective classes, methods, and packages containers. Specifically this filtering is required to minimize the final model in-memory size

[1] http://www.eclipse.org/MoDisco/.

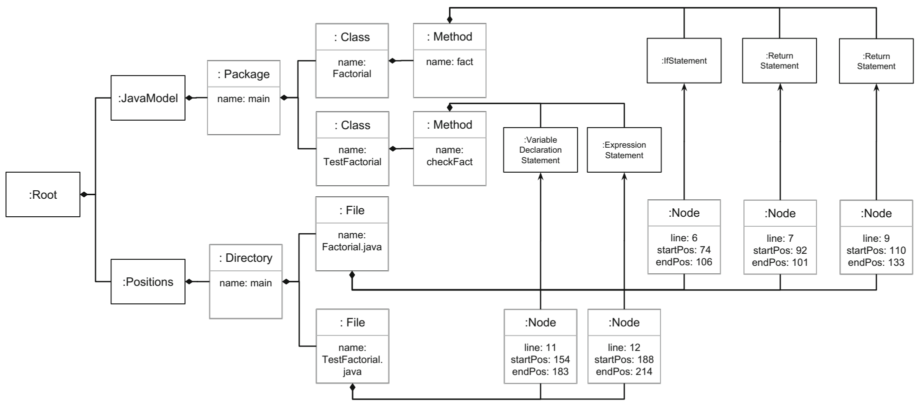

Fig. 3. Excerpt of the static model generated by MoDisco from the code in Fig. 2

afterwards. The filtering is performed during the model transformation described in the next section.

2.2 Model Transformation

The goal of this model transformation is to produce a simpler model to use, containing only the data needed for the program analysis. Moreover, this model must be able to associate methods to statements in order to highlight all the methods which execute a specific statement. This link is created during the execution, i.e. dynamically.

The input of this model transformation is the model statically generated by MoDisco, while the target is a model conforming to a meta-model we introduce. This meta-model needs to differentiate the test classes from the SUT classes (targets), since the execution trace is only computed for the System Under Test. This meta-model is illustrated in Fig. 4.

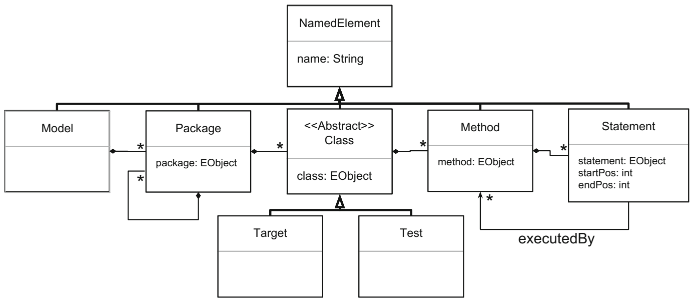

Fig. 4. The impact analysis meta-model.

The `model` element corresponds to the root of the model. It contains packages, which can contain either other packages or classes. Each class can have several methods, which contain a list of statements. Finally, each statement owns a list of references to methods, called `executedBy`. When the model is created by the transformation, this list is empty, nonetheless the completion step described in Fig. 1 consists into filling this list in order to get an execution trace. Since this trace only concerns the SUT classes, the statements of the test suites are not necessary in the analysis model. Thus, target and test classes are differentiated, and only the SUT's statements are kept.

To separate the target classes from the test classes, we assumed that test classes either use the `@Test` annotations, or inherit the `TestSuite` class, from the well known Java testing framework JUnit[2]. That information can easily be accessed inside the static model. Though, several other conditions can be added to the transformation, in order to differentiate the tests from other classes, using the TestNG[3] annotations for example.

The model transformation is made using ATL[4], with the EMF Transformation Virtual Machine (EMFTVM). Indeed, besides including all the standard ATL VM features, EMFTVM offers better model loading and saving, smaller file results, and an overall performance improvement during the ATL transformation [12].

To focus a bit more about the transformation itself, 7 ATL rules are applying a top-down approach on the static model, starting by transforming the packages, classes, and going down to the statements. Apart from the `namedElement`, each element of the meta-model corresponds to an ATL rule.

This transformation drastically reduces the size of the model manipulated in the further sections. In fact this transformation is used to generate the analysis model out of the JUnit4 source code static model. The size of one model is lowered from 50 MB to 1 MB for instance, by keeping only the relevant objects for future processing.

Once this model is built, we need to run the tests in order to produce the dynamic analysis. Before, an instrumentation of the code is necessary, to be able to observe and reify its behavior into the dynamic model during the last step (Sect. 2.4).

2.3 Code Instrumentation

Code instrumentation is conducted by inserting additional statements in the code, at specific places, so that when the SUT is executed, those new statements are executed too [6].

Several instrumentation approaches exist:

– Source code instrumentation consists in adding code into the source file, before compiling it.

[2] http://junit.org/junit4/.
[3] http://testng.org/.
[4] https://eclipse.org/atl/.

– Binary instrumentation modifies or re-writes the compiled code.
– Bytecode instrumentation performs tracing within the compiled code.

During the development of our process, we considered those strategies, and found that the source code instrumentation is the most accurate approach for producing execution traces. Indeed, required information about the source cannot be obtained when analyzing the binaries or the bytecode, especially the source code position of the statements. Besides having the line number for each statement like the other approaches, source code instrumentation can also provide column position numbers. This data will be used to match the executed statements of the code with the analysis model's statements. It is mandatory for the next step (Sect. 2.4) to be able to add the behavior into the static model.

The instrumentation has been performed using the Spoon Framework [9]. This step is done first by iterating over the analysis model, in order to get both Test and Target classes. When instrumenting Test classes, a `setMethod()` statement is inserted at the beginning of each test method. Its parameters are the test class name, and the method name. When instrumenting SUT classes, a `match()` call is added before each statement. Its parameters are the SUT class qualified name where it belongs, the method, and finally the source code position of this statement. The source-code instrumentation generates the code of the Fig. 5, which is compiled and loaded before the execution.

```java
package main;
public class Factorial {
   public int fact(int n) {
      match("Factorial", "fact", 74, 106);
      if (n == 1) {
         match("main.Factorial", "fact", 92,
               101);
         return 1;
      }
      match("main.Factorial", "fact", 110,
            133);
      return n * (fact((n - 1)));
   }
}
```

```java
package main;
public class TestFactorial {
   @Test
   public void testFactorial() {
      setMethod("main.TestFactorial",
            "testFactorial");
      Factorial f = new Factorial();
      Assert.assertEquals(1, f.fact(1));
   }
}
```

Fig. 5. Instrumented code of Factorial class, and its incomplete test suite.

2.4 Execution and Completion

The execution part consists of running the test cases. When the new instrumented statements are executed, the model statically built is completed with dynamic information. Before the test execution, a singleton class is instantiated. The method `setMethod()` sets the current test method being executed into the singleton class.

The other method, `match()`, iterates over the analysis model to find the SUT statement being executed, using its qualified class name, method name, and finally the source code position of the statement. Finally, the `executedBy` list of the newfound statement is completed dynamically, by adding the singleton's current test method. This model dynamically completed reifies the link between statements and test methods, as showed in Fig. 6. Besides showing the test methods that would be impacted when modifying a SUT statement, an empty `executedBy` list shows an uncovered statement, which possibly implies an incomplete test suite. Thus, this model answers to the problematic announced, as it is dynamically created, and analyze the program's behavior i.e. with impact analysis.

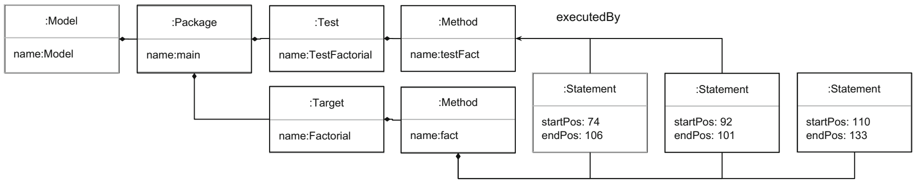

Fig. 6. The analysis model obtained after reifying the link between statements and test methods.

However this model highly depends on the source code's size. Indeed, each statement in the SUT corresponds to one element in the analysis model, thereby an important source code might lead to scalability issues. This scalability is evaluated in the next section.

3 Evaluation

3.1 Execution Environment

In this section we evaluate the overall performances of our dynamic model generation framework, using the XMI persistence layer for our models.

This evaluation is conducted on a simple Java project containing the classes introduced in the Sect. 2. We programmatically increase the size of this project by duplicating the number of classes and test classes. This way, every test class is testing a single target class. Using this setup, we can manage the size of the output model, and observe the behavior of our prototype with either small and big models.

The experiments are executed on a laptop running Windows 7 Professional 64 Bits, using an Intel Core i7-4600 (2.70 GHz) CPU and, a Samsung SSD 840 EVO 250 GB. The Java Virtual Machine version is JDK 1.8.0_121, and runs with a maximum Java heap size of 2,048 MB.

This experimentation starts by running the program analysis on Java Projects containing a few hundred classes, which can be considered as small here. Subsequently, this number of classes is increased, up to thousands. We measured the execution time for each step of our prototype, and reported it in the Fig. 7. As described in the previous parts, those steps are: Reverse Engineering (RE), Model Transformation (ATL), Source code Instrumentation (Instr), Test Execution (Exec).

When the project under analysis reaches approximately 12,000 test classes, the model created by MoDisco using reverse engineering is too big to be stored in memory, thus preventing any other analysis on bigger projects.

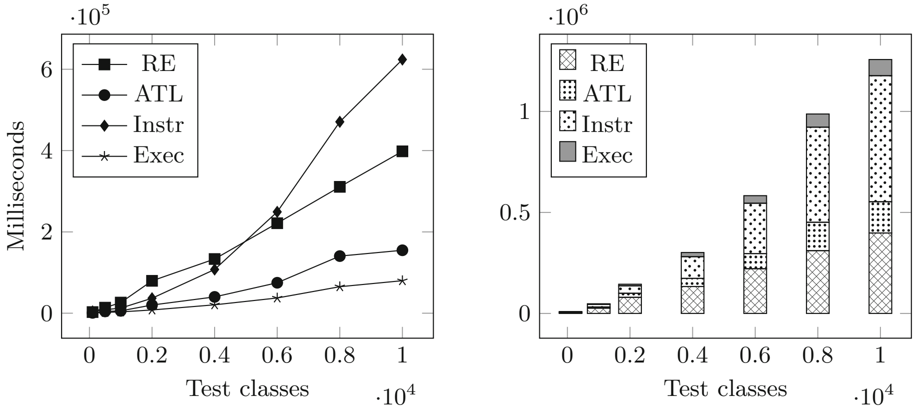

Fig. 7. Dynamic program analysis execution times using the XMI persistence layer.

3.2 Discussion

The curves from the left diagram in the Fig. 7 are showing the growth of execution times when the number of classes increases. The other diagram shows the same data, but its representation gives a better understanding of each step's duration in the whole process and its total duration. This diagram shows that the MoDisco static model generation, and the source code instrumentation are by far the longest steps of the program analysis, up to 32% for the reverse engineering step, and 50% for the instrumentation step, approximately.

This can be explained by the fact that both are parsing the whole source code. Yet it is interesting to notice that the instrumentation tends to be longer than the reverse engineering once the model contains more than 5,000 test classes. The reason is the Spoon's instrumentation's complexity being more than linear.

Also, as written in the previous subsection, the MoDisco static model creation will not be achieved when the program under analysis gets very big (approximately 12,000 test classes) due to a lack of memory and the well-known XMI scalability problem.

Pagán et al. explained in their paper [8] that the XMI persistence layer scales badly with large models, due to the fact that XMI files cannot be partially loaded. Indeed the XMI resource needs to keep the complete object in the memory to use it.

This scalability problem can be partially resolved using a more scalable persistence layer for the EMF Models, such as NeoEMF [2] or CDO[5]. Nonetheless, MoDisco has its own meta-models, and uses EMF generated code. Using this code with NeoEMF and CDO resources cannot currently improve the scalability, since those layers need to generate their own code from an Ecore meta-model. It is one of our future challenges.

4 Related Work

In order to reduce the maintenance costs of large applications, knowing the impacts a code modification can have on other parts of the program can really improve the developers life. Several approaches already exist within the impact analysis domain.

PathImpact [7] is a dynamic impact analysis technique that does not require any static analysis during its process. PathImpact instruments the binaries or a program, and generates traces during its execution. This execution trace is then used to create call graphs, which are analysed in order to study the impact analysis. The impacts are identified at the method level, a coarser level than our approach based on source code instrumentation.

Chianti [10] is a change impact analysis tool based on call graphs. Using two different versions of the source code, Chianti creates a set of *atomic changes*, and analyses their dependencies in order to determine the tests affected by code changes. Basically, considering two atomic changes A_1 and A_2, if adding A_1 in the first version of the source code leads to a program syntactically invalid, then it has a dependency to A_2. For each tests, a call graph is generated (statically or dynamically), and from those dependencies, operations affecting the tests can be identified. This approach is really different from ours since it needs two version of the source code. This impact analysis are more coarse-grained than the approach presented in this paper, i.e. classes, methods and fields, but Chianti supports more operations, such as insertion and deletion.

Imp [1] is a static impact analysis tool for the C++ language based on program slicing [13]. Program slicing consists into computing a set of points, such as statements, that can have an effect on other point of the program. To compute the impact analysis, Imp only considers “forward” slicing, which computes a set of statements that are affected by a previous point. This approach is fine-grained, since the analysis can be done at the statement level. Nonetheless it suffers from the disadvantages of the static approach, a loss of precision, in order to limit the execution time when the program’s source code is being too big to analyse.

5 Conclusion and Future Works

In this paper we present our prototype for dynamic program analysis purposes, using Model Driven Engineering. The multiple steps of the process dynamically

[5] https://eclipse.org/cdo/.

generate a model, highlighting the links between program's statements and test suites. Nevertheless, the experimentations led on this prototype showed that it suffers from the XMI scalability issues induced by the MoDisco static analysis.

Our future work will focus on the scalability of our prototype, more specifically by integrating a more efficient persistence layer in term of scalability, such as CDO or NeoEMF. Furthermore, the source code instrumentation remains a very cumbersome technique, and evolving towards a more common solution, such as "on the fly" byte-code instrumentation would be interesting, especially if the granularity is maintained.

Acknowledgment. This work is supported by DIZOLO project - Aurora mobility programme 2017.

References

1. Acharya, M., Robinson, B.: Practical change impact analysis based on static program slicing for industrial software systems. In: Proceedings of the 33rd International Conference on Software Engineering, pp. 746–755. ACM (2011)
2. Benelallam, A., Gómez, A., Sunyé, G., Tisi, M., Launay, D.: Neo4EMF, a scalable persistence layer for EMF models. In: Cabot, J., Rubin, J. (eds.) ECMFA 2014. LNCS, vol. 8569, pp. 230–241. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-09195-2_15
3. Bézivin, J.: On the unification power of models. Softw. Syst. Model. **4**(2), 171–188 (2005)
4. Bruneliere, H., Cabot, J., Dupé, G., Madiot, F.: Modisco: a model driven reverse engineering framework. Inf. Softw. Technol. **56**(8), 1012–1032 (2014)
5. Ernst, M.D.: Static and dynamic analysis: synergy and duality. In: ICSE Workshop on Dynamic Analysis, WODA 2003, pp. 24–27 (2003)
6. Gosain, A., Sharma, G.: A survey of dynamic program analysis techniques and tools. In: Satapathy, S.C., Biswal, B.N., Udgata, S.K., Mandal, J.K. (eds.) Proceedings of the 3rd International Conference on Frontiers of Intelligent Computing: Theory and Applications (FICTA) 2014. AISC, vol. 327, pp. 113–122. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-11933-5_13
7. Law, J., Rothermel, G.: Whole program path-based dynamic impact analysis. In: 2003 Proceedings of the 25th International Conference on Software Engineering, pp. 308–318 (2003)
8. Pagán, J.E., Cuadrado, J.S., Molina, J.G.: Morsa: a scalable approach for persisting and accessing large models. In: Whittle, J., Clark, T., Kühne, T. (eds.) MODELS 2011. LNCS, vol. 6981, pp. 77–92. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24485-8_7
9. Pawlak, R., Monperrus, M., Petitprez, N., Noguera, C., Seinturier, L.: SPOON: a library for implementing analyses and transformations of Java source code. Softw. Pract. Exp. **46**(9), 1155–1179 (2016)
10. Ren, X., Shah, F., Tip, F., Ryder, B.G., Chesley, O.: Chianti: a tool for change impact analysis of Java programs, vol. 39, no. 10, pp. 432–448 (2004)
11. Sherwood, K.D., Murphy, G.C.: Reducing code navigation effort with differential code coverage. Department of Computer Science, University of British Columbia, Technical report (2008)

12. Wagelaar, D., Tisi, M., Cabot, J., Jouault, F.: Towards a general composition semantics for rule-based model transformation. In: Whittle, J., Clark, T., Kühne, T. (eds.) MODELS 2011. LNCS, vol. 6981, pp. 623–637. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24485-8_46
13. Weiser, M.: Program slicing. In: Proceedings of the 5th International Conference on Software Engineering, pp. 439–449. IEEE Press (1981)