ProFit – Performing Dynamic Analysis of Software Systems

Antonio García de la Barrera(✉), María Ángeles Moraga, Macario Polo, and Ignacio García-Rodríguez de Guzmán

Alarcos Research Group, Institute of Technologies and Information Systems, University of Castilla-La Mancha, Ciudad Real, Spain
antonio.garcialabarrera@alu.uclm.es, {mariaangeles.moraga, macario.polo, ignacio.grodriguez}@uclm.es

Abstract. Dynamic analysis offers the possibility of studying software at runtime, documenting its internal behavior. This dynamic information about the software is very interesting for the purpose of identifying many aspects of its operation, such as detection of dead code, security problems, complexity, and so on. However, not all software systems have the capacity to generate detailed information about what happens at runtime. It is with that consideration in mind that in this work we present ProFit, an environment conceived to improve software with the capacity to generate dynamic information about its execution, thus complementing the static analysis that can be performed on it. ProFit implements two strategies for such purpose: (i) instrumentation of the source code, through the insertion of sentences that generate execution traces in log files, and (ii) automatic generation of aspects for the generation of execution traces. None of those strategies produces any alteration in the behavior of the software, so the information generated truly reflects what happens during the software execution. Finally, the execution logs are represented by means of a tree-like structure that makes it quite easy to implement several kinds of analysis on it.

Keywords: Dynamic analysis · Software maintenance · Execution trace · Instrumentation · Aspects

1 Introduction

The Quality of software can be assessed from multiple perspectives [1], obtaining measurements that determine the level of Quality in dimensions such as usability, maintainability, sustainability, performance, etc. [2].

Profiling (analysis of the behavior of the software at runtime) has thus been used not only as a tool to improve the understanding of the systems when performing reverse engineering, but also in the analysis of the energy consumption of the software [3], in the calculation of the degree of coverage in the processes of testing [4], in the detection of bottlenecks hindering the improvement of the performance [5] or in functionality identification of the System Under Analysis (SUA) [6].

M. Piattini et al. (Eds.): QUATIC 2019, CCIS 1010, pp. 255–262, 2019.
https://doi.org/10.1007/978-3-030-29238-6_18

In an effort to assess software system Quality in several dimensions, we are currently developing a toolbox for source code dynamic analysis, made up first of all of a tool (called ProFit) that makes it possible to generate an instrumented copy of a given System Under Analysis. The copy preserves the semantics of the original SUA but generates execution traces at runtime.

Secondly, we will implement a set of tools which will analyze the quality in several dimensions, using the traces left by the instrumented SUA in different executions as input, as illustrated in Fig. 1.

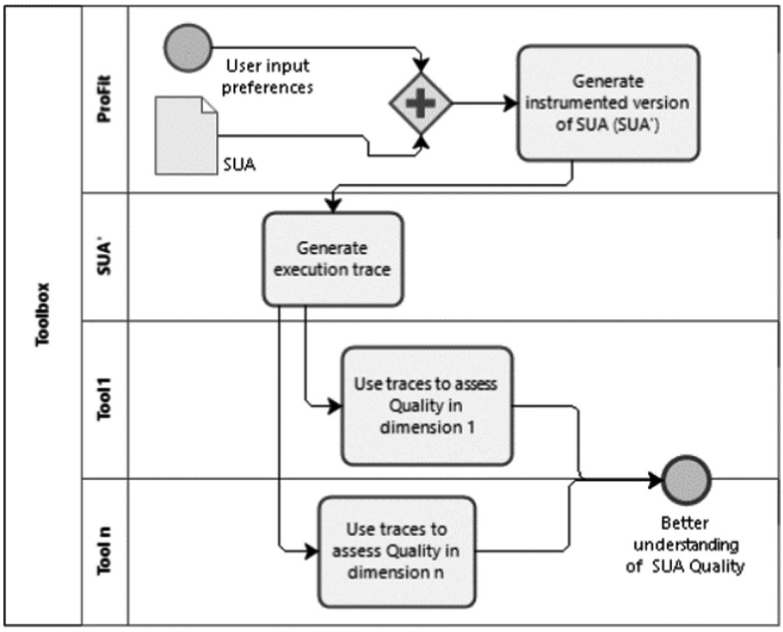

Fig. 1. Proposed toolbox workflow.

In this paper we present ProFit, a profiling tool for Java code that offers the researcher or developer the possibility to instrument the code automatically, allowing a choice of whether the instrumentation is done by the insertion of instructions in the source code, or through the addition of Aspect-oriented Programming (AOP). In addition, the tool enables the analysis options to be parameterized, making it possible to select which components of the system should be analyzed, and what information should be obtained from each one.

The paper is organized as follows: Sect. 2 sets out a general overview of the most important concerns; namely, code instrumentation, object-oriented aspect technique, and existing proposals; Sect. 3 presents the general view of the architecture of ProFit; finally, Sect. 4 outlines several conclusions and ideas as regards future work, as well providing details about the ongoing validation.

2 Background

This section will present the basics of ProFit (source code instrumentation and Aspect-oriented Programming).

2.1 Code Instrumentation

Instrumentation consists in the addition of instructions to the code of a system in such a way that, without modifying its behavior from a functional perspective, traces of execution can be obtained that inform the engineer in great detail of what is happening at any given time. There are many possible purposes for this instrumentation, such as to count the number of cache failures, to calculate the time (and/or energy) invested in a specific region of the code, to study the parameters given to a function, or the values that a variable contains throughout the execution, to name but a few examples [7].

2.2 Aspect-Oriented Programming

AOP has been used extensively to encapsulate non-business, transversal functionalities (like logging, authentication, transactions, etc.) which compromise several domain classes through different object-oriented modules.

"The main idea behind AOP is to consider these operations as crosscutting concerns and to group them into separate modules, which are automatically called by the system" [8]. In the previous state of the art, AOP has been found useful in software Quality assessment [9].

In regard to the present paper, AOP is used as a way of introducing code instructions in the SUA workflow without any direct modification of the source code, thus creating an alternative to direct source code insertions on profiling.

3 Architecture of the Proposal

In this section, the architecture and functionality of the technological framework will be discussed. Firstly, the overall structure will be presented, by a discussion of the different modules and the advantages of the architecture.

3.1 Overall Structure

The proposed architecture faces two challenges which are common to this kind of tools: the extensibility (of source code languages and techniques to instrument) and flexibility (the capacity to configure the instrumentation so as to produce customized execution information).

The main criterion behind the design is scalability. ProFit's design aims for a modularity that enables there to be consistent and ongoing addition of new instrumentation techniques or languages.

On one hand, in order to add the capability of analyzing systems in different languages, a new file for these must be implemented in the analyzer, instrumenter and aspectsGenerator modules, and some additions to the InstrumentationPerformer class need to be made. This is, however, limited to object-oriented languages.

On the other hand, in order to increment the number of instrumentation techniques that ProFit can perform, a new module must be developed. This module containing the technique logic, in conformity with existing interfaces, is placed alongside the already-existing ones.

The current version of ProFit deals with Java, as well as with two techniques to provide SUAs with the execution trace generation functionality: (i) source code insertion and (ii) Aspect-oriented Programming (Fig. 2).

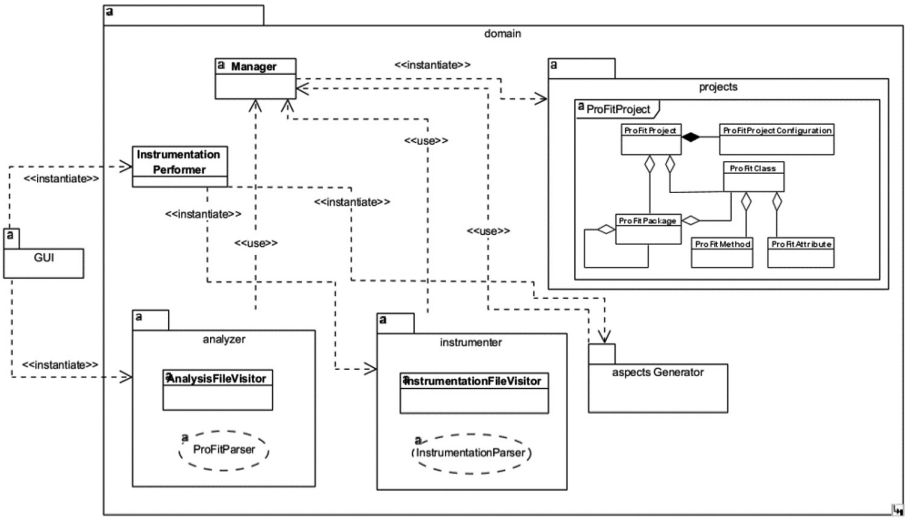

Fig. 2. ProFit overall architecture.

3.2 A Suitable Graphic User Interface

ProFit has a Graphic User Interface (GUI) that allows the user to select the SUA; it shows its structure through a dropdown tree, enabling the user to choose which parts are subject to analysis, as well as to determine a set of "pattern-*statements*". Any method whose name contains one of those *statements* will thus be automatically included in instrumentation.

The GUI also makes it possible for the user to select the parameters which will determine the nature of the instrumentation, namely:

- Regarding attribute operations traces: the user can either write the code to insert directly, or let it be autogenerated, choosing if parameter values are going to be included in the trace.
- Regarding method call traces: the user can either write the code to insert directly, or let it to be autogenerated, choosing if timestamps must be included.
- Regarding output format: the user can choose one or more of the following output formats: console output, .txt file, .csv file and serialized file.

3.3 Analyzer Module: Acquiring Knowledge from SUA

The first task in performing any instrumentation technique is to analyze the Legacy System architecture, so as to acquire the necessary knowledge for the subsequent actions.

For this purpose, ProFit first analyzes the SUA structure recursively, exploring its package and class composition. When a class is found, the system uses a modified parser to explore its code, obtaining the attributes and methods it contains.

Code 1 displays an example class, HelloWorld, designed to illustrate the output of the different instrumentation techniques discussed below.

```
public class HelloWorld {
        static String aString;
        public static void main( String args[] ) {
                salute();
        }
        private static void salute() {
                aString = "Hello world!";
                System.out.println(aString);
        }
}
```

Code 1. HelloWorld.java

3.4 First Strategy: Source Code Insertion

The direct source code insertion technique is based on the use of a second Java parser, created by extending the Java language grammar, as well as on a state machines module.

```
import traceWriter.TraceWriter;

public class HelloWorld {
        static String aString;
        public static void main( String args[] ) {
                salute();
        }
        private static void salute() {
TraceWriter.writeTxt( new MethodCallTrace( /*...*/);
TraceWriter.writeTxt( new FieldSetTrace( /*...*/);
                aString = "Hello world!";
TraceWriter.writeTxt( new FieldSetTrace( /*...*/ );
                System.out.println(aString);
TraceWriter.writeTxt( new MethodCallTrace( /*...*/);
        }
}
```

Code 2. Example class after code insertion.

The workflow is as follows: the parser reads a Java file and from a grammatical perspective identifies the parts of the code which are potentially subject to code

insertions related to methods (also specific methods specified in the configuration of the execution trace). From a lexical perspective, those parts which are potentially subject to attribute modification-related code insertions are identified. Then the state machines build, if appropriate, the code line to be inserted.

3.5 Second Strategy: Aspect-Oriented Instrumentation

Aspect-oriented Programming allows cross-cutting functionalities to be added to an already existing system, without any modification to the original code. This is achieved by means of creating a new, autogenerated file, determining the interesting pointcuts in the original code, as well as the behavior when these are reached.

This is implemented by the execution of a grammar file, which, taking the SUA information and user's preferences as inputs, builds a file called *InstrumentationAspect.aj,* and inserts it into the SUA.

```
public aspect InstrumentationAspect {
        public pointcut DynamicMethodCall(Object object) : (
                ( call(* HelloWorld.salute(..)) && target(object))
        );
        public pointcut AttributeOperation() : (
                ( set(* HelloWorld.aString))
        );
        before(Object object) : DynamicMethodCall(object) {
                MethodCallTrace methodCall = new
        MethodCallTrace(/*...*/);
                TraceWriter.writeTxt(methodCall);
        }
        // [...]
}
```

Code 3. Aspect file autogenerated for the example class.

The source code presented in **Code 3** displays an example of the generated file, taking as inputs the example file, HelloWorld.java, and the selection of the attribute "*String aString*" and the method "*void salute()*" by the user, who also determines the log to be stored as a.txt file.

3.6 Management of the Persistence of the Execution Traces

ProFit offers the user four different log output file types, namely: console output, .txt file, .csv file and serialized.txt.

The first three come in a more verbose, human-readable format, similar to JSON[1] (but without nested values, and thus incomplete). Serialized output is a non-human-readable, but recursive (and thus complete) format, designed to be loaded and visualized on a work-in-progress trace analysis system.

[1] https://www.json.org/.

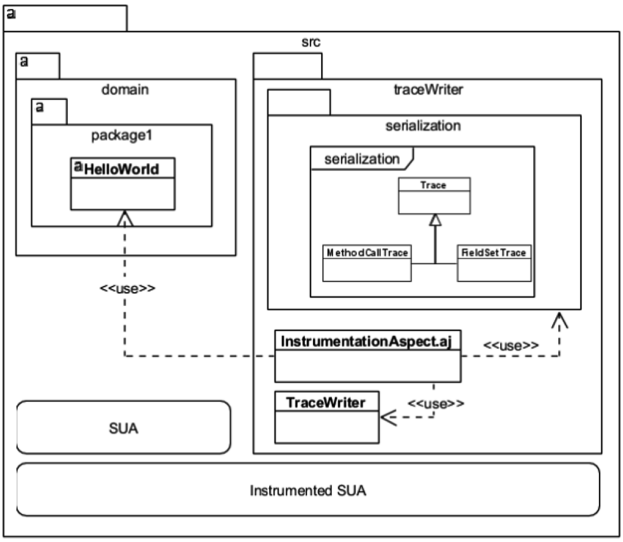

Fig. 3. SUA structure after aspect-oriented instrumentation.

In order to achieve this, a new package is inserted into the SUA, containing a class, called TraceWriter; this class, with a single instance, receives all the traces from the SUA and creates the files necessary to execute the persistence.

In the same package, moreover, a data structure is added, in order to represent the traces, as illustrated in Fig. 3.

4 Conclusions

This paper presents ProFit, a profiling tool to improve software systems developed in Java, with the capability of generating configurable execution traces. ProFit implements two strategies for profiling: (i) source code instrumentation and (ii) use of aspects. The result is a functionally equivalent software system that generates information about its execution in the form of log files when it runs.

This is verified by means of the instrumentation of a system with a 95.5% coverage test suite, and the execution of both the original system tests and those of the instrumented one, checking that the test results remain the same. The log created by the instrumented system during tests is also evaluated, verifying that it properly contains the required information about the system execution.

Regarding future (and ongoing) work, these traces will later be transformed into an execution tree that will be used to perform various types of analysis related to the measurement of some of the dimensions of Quality.

Case studies are currently being carried out both on open-source systems and on tools provided by companies, seeking to identify Quality defects that can be detected only at runtime.

Acknowledgements. This work has been funded by the TESTIMO and SOS projects (Consejería de Educación, Cultura y Deportes de la Junta de Comunidades de Castilla La Mancha, y Fondo Europeo de Desarrollo Regional FEDER, SBPLY/17/180501/000503 and SBPLY/17/180501/000364, respectively).

References

1. ISO: ISO/IEC 25000 - Requisitos y Evaluación de Calidad de Productos de Software (SQuaRE - System and Software Quality Requirements and Evaluation), International Organization for Standarization, Ginebra (2005)
2. ISO: ISO/IEC 25040. Systems and Software Engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – Evaluation Process, International Organization for Standarization, Ginebra (2011)
3. Jagroep, E.A., et al.: Software energy profiling: comparing releases of a software product. In: Proceedings of the 38th International Conference on Software Engineering Companion, New York, NY, USA, pp. 523–532 (2016)
4. Tikir, M.M., Hollingsworth, J.K.: Efficient instrumentation for code coverage testing. In: Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis, New York, NY, USA, pp. 86–96 (2002)
5. Shen, D., Luo, Q., Poshyvanyk, D., Grechanik, M.: Automating performance bottleneck detection using search-based application profiling. In: Proceedings of the 2015 International Symposium on Software Testing and Analysis, New York, NY, USA, pp. 270–281 (2015)
6. Del Grosso, C., Di Penta, M., de Guzman, I.G.-R.: An approach for mining services in database oriented applications, pp. 287–296 (2007)
7. Tirado-Ramos, A., et al.: Computational Science – ICCS 2009: 9th International Conference Baton Rouge, LA, USA, May 25-27, 2009 Proceedings, Part II, 1st edn. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-01973-9
8. Jacques Pasquier, P.F.: Mini-proceedings of the master seminar advanced software engineering topics: aspect oriented programming. University of Fribourg (2006)
9. Soni, G., Tomar, P., Upadhyay, A.: Analysis of software quality attributes through aspect-oriented programming (2013)