A Proposal for an Extensible Benchmark of Execution Traces

Raphaël Khoury[1], Sylvain Hallé[1], and Abdelwahab Hamou-Lhadj[2]

[1] Department of Mathematics and Computer Science Université du Québec à Chicoutimi
raphael.khoury@uqac.ca, shalle@acm.org
[2] Department of Electrical and Computer Engineering
Gina Cody School of Engineering and Computer Science
Concordia Universiry
wahab.hamou-lhadj@concordia.ca

Abstract

In this position paper, we propose a benchmark for the testing of runtime monitors and other trace analysis tools, with a special focus on execution traces from Java programs. The proposed benchmark uses traces generated using AspectJ, thus offering the benefit that any researcher is able to generate new traces in the same format, using his\her own Java programs. We argue that these traces contain sufficient information to detect several behaviors of interest in program execution, and illustrate this fact with examples.

1 Introduction

Execution traces form an essential basis for several types of program analysis and software engineering tasks including debugging, feature enhancement, performance analysis, and their usefulness has in turn let to a proliferation of verification tools, that allow users to verify the compliance of traces with properties stated in widely differing formalism.

This diversity of formalism has made it difficult to compare all tools on an equal footing. In this position paper, we propose an benchmark, consisting of five traces and several properties that can be verified over these traces, that we believe form an adequate benchmark for runtime verification. Indeed, we argue that an adequate trace benchmark should meet the following requirements.

Objectives We argue that an adequate trace benchmark should meet the following requirements.

1. The traces must be easy to generate, so that any researcher can generate new traces in the same format, and thus add to the repository.
2. Versatility: reading the trace should necessitate no specialized program. In particular, the trace must be human readable with a certain level of ease.
3. Completeness: The trace should contain sufficient information about the execution of the underlying program to diagnose most common software bugs and security vulnerabilities as well as to generate useful profiling data about the program runtime performance.
4. Customizable: The trace format should be completely customizable, allowing users to suppress any information that is extraneous to their intended use. Furthermore, easy customization of the output format is necessary when the monitor requires input traces in a specific format, and does not itself allow for customization of its input format.

We claim that the benchmark proposed in this position papers meets each of the above requirements.

In particular, the trace is generated using AspectJ [4], a tool that allows adding executable blocks to the source code without explicitly changing it. In our case, we used AspectJ to insert code before and after every method call to record the information needed to perform monitoring and have made available alongside with our benchmark. This choice means that users can readily generate new traces for their own Java applications, as well as modify the format of the traces to fit their monitor. Finally, as can be seen in Section 3, this trace format contains sufficient information to verify interesting properties about programs.

At present, the benchmark contains the following traces:

Trace Name	Source	num of lines	Size (MB)
(1)TetrisTrace	A freely available Java Tetris game	10 457 631	763
(2)ChatTrace	Several instant messaging conversations	11 478	2,67
(3)Semaphore	A semaphore manipulation tutorial program	1,429	0,148
(4)FractalTree	The drawing of a Fractal Tree using a graphics editor	1 545 499	375
(5)Encription	The encryption of a file using AES	1 476 228	138

Table 1: Currently available traces

Each of these traces is generated using freely available Java Code. Links to each program is provided in the benchmark's associated README.txt file. As mentioned above, the benchmark is easily extensible, using the AspectJ script provided alongside with the benchmark, and we expect other traces to be added by ourselves and other researchers in the near future. The benchmark is freely available at: https://github.com/RacimBoussaha/TraceSamples.

2 Proposed Traces format

Each line correspond to either a single method call or method return. A *call* line is added to the trace whenever a method is called, with the first line of the trace naturally corresponding to the initial call to the main function[1]. A corresponding *return* line is added when that function returns. Each line begins with a timestamp, followed by a unique identifier that links a call to the corresponding return and by the keyword 'call' or 'return', identifying the line's type. In case of call line, it will then list the following information, separated by comas: the method's containing class or interface and its package, a hashcode of the calling object, the methods return type, its name, its nesting depth, the number of parameters, a list of parameter types (separated by // ') and a similarly formatted list of parameter values. A return method lists only the return type and value.

Values of literals, string and elementary types are provided explicitly but those of objects are provided by references, allowing tracking of data objects across the execution. Arrays are prefixed with '[' and construction are suffixed with<init>. The ids of objects and method calls are computed by AspectJ. More information about the structure of the traces is provided in the accompanying README.txt file.

A sample of the trace, with timestamps and method ids elided out of space considerations, is given in Figure 1. The complete benchmark (traces and the generating code) is available at: https://github.com/RacimBoussaha/TraceSamples.

[1] Actually, dependencies of the program, some initialization might occur before main is called. In the sample below, main is the third line, after a constructor is called and returns.

```
call,NA,NA,void,SemaphoreTest.main,0,1,[Ljava.lang.String;,[Ljava.lang.String;@b4c966a
call,NA,NA,void,java.lang.System.out,1,0,NA,NA
return,java.io.PrintStream@4e50df2e,0,NA,NA,NA,NA,NA,NA
call,NA,NA,void,java.lang.StringBuilder.<init>,1,1,java.lang.String,Total
available Semaphore permits :
return,Total available Semaphore permits : ,0,NA,NA,NA,NA,NA,NA
```

Figure 1: A fragment of a trace

3 Sample Properties

In this section, we propose a selection of real-life properties that can be verified with traces of the format described above. As can be seen by these examples, the proposed trace format is sufficiently expressive to allow the verification of complex, fine-grained properties on program behavior. We have also opted to incorporate into this benchmark properties stated in multiple different formalisms, thus illustrating the versatility of the dataset. Indeed, it is not clear if there exists a single specification language capable of stating everyone of the properties listed below, and for one specific property (prop2), we were unable to provide formal definition. The table 2 shows the result of testing the properties above on the traces in our benchmark.

Property	trace	verdict
prop1	TetrisTrace	T
prop2	FractalTree	F
prop3	ChatTrace	T[2]
prop4	ChatTrace	?
prop5	Encryption	T
prop6	Encryption	F
prop7	ChatTrace	T
prop8	Semaphores	T

Table 2: Properties

Profiling and Program Behavior The trace is sufficiently complete to reconstruct the control flow of the program. indeed, in [1], we use similar traces to produce a graph highlighting, for each method call in the trace, the number of times it directly calls every other method. The benchmark contains two properties of this type. Prop1 verifies the correct behavior of the Tetris program, and imposes a new piece may spawn only if the previous piece has stopped falling. It is given in LTL as : $\phi\neg\mathbf{F}\,(\neg\texttt{hasFinishedFalling}\wedge\texttt{newPiece})$. On trace 4, we can verify a more involved property (prop2). Early in the trace, a window panel is created through a method `JFrame.panel()`, whose 3rd and 4th parameters indicate the size of the panel. We must then verify that the program never writes outside these boundaries using the `OnDrawLine()` method.

Numerical Properties Several researchers have recently expressed a growing interest in providing a quantitative, rather than qualitative verdict, on the evaluation of a specification, or security policy on traces. Khoury et al. [3] propose an extension of LTL that such numerical verdicts to be expressed. A typical query in TK-LTL could be: How many times is the method `Java.Random` called in this trace? Such numerical properties can be put in boolean form by comparison, for example: 'Is there a point in the trace at which more calls to `method1` have occurred than calls to `method2`'. On trace 2, an interesting property is to verify that there is never more than N clients connected to the server simultaneously. This property is given in TK-LTL as (prop3): $\phi\forall_{<N}(\mathcal{C}^{\top}_{\texttt{connect}} - \mathcal{C}^{\top}_{\texttt{disconnect}})$

Even more elaborate properties can be stated. The return from a call to `StringBuilder` contains the string that has been built. Computing the sum of the length of these strings is useful for optimization purposes(prop4).

IsKey \IsNotKey isKey and isNotKey are a pair of program analysis properties that illustrate the expressiveness of the approach since few security property enforcement languages are sufficiently expressive to state both properties [5]. The first property states that a given piece of information provided in the trace, such a parameter to a specific method or its return value, must never take the same value twice. Conversely, the isNotKey property imposes that any such value occurs in a matching pair. The tool allows users to specify the trace element to which they wish to apply the property. We verify both properties on trace 4, by checking that the same file is never encrypted twice (prop5-6).

Classical Properties We included in the benchmark two more common properties. Trace 2 makes use of an iterator, and thus must respect the common `next()-hasNext()` property. Trace 3 is a program that manipulates semaphores, allowing us to state and verify a `SafeLock()` property. See [2] for an automata representation of both properties (prop7-8).

4 Conclusions

In this position paper, we propose a benchmark of Java execution trace, for use un runtime verification. The benchmark's main benefit is that it is easily extensible, since the AspectJ code that generates these traces is freely available. The traces are sufficiently detailed to verify an number of properties of interest and we have deliberately included in the benchmark properties from a wide varieties of formalisms thus illustrating the versatility of the dataset.

References

[1] Abdessamad Imine, José M. Fernandez, Jean-Yves Marion, Luigi Logrippo, and Joaquín García-Alfaro, editors. *Foundations and Practice of Security - 10th International Symposium, FPS 2017, Nancy, France, October 23-25, 2017, Revised Selected Papers*, volume 10723 of *Lecture Notes in Computer Science*. Springer, 2018.

[2] Dongyun Jin, Patrick O'Neil Meredith, Choonghwan Lee, and Grigore Roşu. Javamop: Efficient parametric runtime monitoring framework. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 1427–1430, Piscataway, NJ, USA, 2012. IEEE Press.

[3] Raphaël Khoury and Sylvain Hallé. Tally keeping-ltl: An LTL semantics for quantitative evaluation of LTL specifications. In *2018 IEEE International Conference on Information Reuse and Integration, IRI 2018, Salt Lake City, UT, USA, July 6-9, 2018*, pages 495–502. IEEE, 2018.

[4] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. *An Overview of AspectJ*, pages 327–354. Springer Berlin Heidelberg, 2001.

[5] Luc Segoufin. Automata and logics for words and trees over an infinite alphabet. In *Computer Science Logic, 20th International Workshop, CSL 2006, 15th Annual Conference of the EACSL, Szeged, Hungary, Sept. 25-29,*, pages 41–57, 2006.