SMock — A Test Platform for Monitoring Tools[⋆]

Christian Colombo[1], Ruth Mizzi[1] and Gordon J. Pace[1]

Department of Computer Science, University of Malta
{christian.colombo | rmiz0015 | gordon.pace}@um.edu.mt

Abstract. In the absence of a test framework for runtime verification tools, the evaluation and testing of such tools is an onerous task. In this paper we present the tool SMock; an easily and highly configurable mock system based on a domain-specific language providing profiling reports and enabling behaviour replayability, and specifically built to support the testing and evaluation of runtime verification tools.

1 Introduction

Two of the major challenges in runtime verification, which are crucial for its adoption in industry, are those of the management of overheads induced through the monitoring and the ensuring the correctness of the reported results. State-of-the-art runtime verification tools such as Java-MOP [7] and tracematches [1] have been tested on the DaCapo benchmark[1], but the kind of properties in these case studies were typically rather low level, contrasting with our experience with industrial partners who are more interested in checking business logic properties (see e.g., [4, 3]). Whilst we had the chance to test our tool Larva [5] on industrial case studies, such case studies are usually available for small periods of time and in limited ways due to confidentiality concerns. Relying solely on such case studies can be detrimental for the development of new tools which need substantial testing and analysis before being of any use.

To address this lack, we have built a configurable framework which may be used to mock transaction[2] systems under different loads and usage patterns. The key feature of the framework is the ease with which one can define different types of transactions and farm out concurrent instances of such transactions through user-specified distributions. Profiling the overheads induced by different runtime verification tools, thus, enables easier benchmarking and testing for correctness of different tools and techniques under different environment conditions. Although not a replacement of industrial case studies, this enables better scientific evaluation of runtime verification systems.

SMock allows straightforward scripting of case studies, giving control over transaction behaviour, timing issues, load buildup, usage patterns, etc., which can be used to benchmark the correctness and performance of different runtime verification tools. A major issue SMock attempts to address is that of repeatability of the experiments,

[⋆] The research work disclosed in this publication has been partially funded by the University of Malta Research Fund 2012, grant number CPSRP04-02.

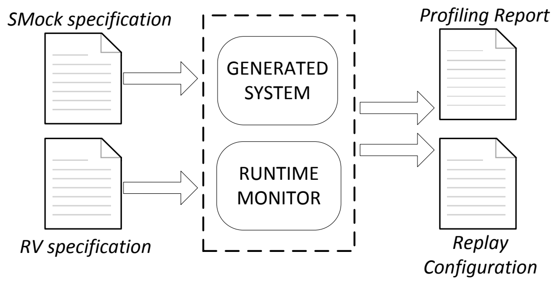
[1] `http://www.dacapobench.org/`

[2] We use *transaction* to refer to an independent, highly replicated unit of processing.

ensuring the scientific validity of the conclusions. To evaluate the use of the tool, we have used it to build a suite of benchmarks and compared the performance of JavaMop [7] and polyLarva [2] under different mocked scenarios.

2 The Architecture and Design of SMock

SMock has been built in such a manner so as to enable the easy generation of families of transaction systems with two levels of configurability: (*i*) the form of the underlying transactions (possibly including variations); and (*ii*) the scenarios depicting transaction firing as distributed over time. To enable us to achieve this goal, SMock takes a script describing these options and generates a Java mock system behaving in the prescribed manner, generating events which can be captured by monitoring tools. Such a system is then typically monitored using a specification script in a runtime monitoring tool, enabling the measurement of overheads induced by the monitors in such a setting. Since the mock system may include parts which are randomly generated (e.g., launch between 100 and 120 transactions in the first 10 seconds), its execution also tracks information to allow exact replayability. This enables, for instance, comparison of the performance of the different versions of a runtime verification tool, or its performance against that of other tools, or as a regression test following a bug fix. The general usage pattern of SMock is shown below:



3 Scripting Behaviours

One of the strengths of SMock is that it allows users to script scenarios to be used for testing. The scripts are written in a small domain-specific language which provides a number of constructors to enable the description of processes and how they are to be launched. Basic processes are individual actions which are characterised by three parameters — their duration, memory consumption and CPU usage. These parameters can be exact values or probability distributions[3], to enable the mock system to behave within a range of possible behaviours. Actions can be named if one would want to be able to monitor their moment of entry and exit, but can also be left unnamed.

Processes can be combined using the following operators: (*i*) sequential composition; (*ii*) probability weighted choice between processes; and (*iii*) parallel composition of processes. The combinators come in a binary form and also generalised versions which allow the combination of multiple processes together.

[3] The tool currently supports uniform and normal distributions.

Example 1. Consider a document management system which allows users to login, browse, upload documents, edit these documents, etc. The resources used by some of these actions can be characterised as follows:

$$\begin{aligned}
\text{login} &\overset{\text{df}}{=} \text{action}\langle \text{duration} = \text{uniform}(3,5), \text{memory} = \text{normal}(20,40), \text{cpu} = 0.1\rangle\\
\text{browse} &\overset{\text{df}}{=} \text{action}\langle \text{duration} = \text{normal}(5,8), \text{memory} = 300, \text{cpu} = \text{normal}(0.5,0.7)\rangle\\
\text{logout} &\overset{\text{df}}{=} \text{action}\langle \text{duration} = \text{normal}(1,3), \text{memory} = \text{normal}(20,40), \text{cpu} = 0.1\rangle
\end{aligned}$$

One can now generate 300 users acting in one of two possible ways — browsing or editing a number of files:

$$\begin{aligned}
\text{usertype1} &\overset{\text{df}}{=} \text{login;}\ \text{browse;}\ \text{logout}\\
\text{usertype2} &\overset{\text{df}}{=} \text{login;}\\
&\quad\ \text{seq foreach document} \in \{1\ldots3\}\ \text{do}\\
&\quad\quad\quad \text{open; edit; save; edit; close;}\\
&\quad\ \text{logout}\\
\text{system} &\overset{\text{df}}{=} \text{par foreach user} \in \{1\ldots300\}\ \text{do}\\
&\quad\quad\quad \text{choice}\begin{cases} 0.9 \mapsto \mathit{usertype}1\\ 0.1 \mapsto \mathit{usertype}2\end{cases}
\end{aligned}$$

In practice, for a more realistic scenario, we would not want the user transactions to be launched all together at a single point in time, so we would add an (unnamed) action preceding each user transaction, which takes some time to terminate, but does not consume CPU or memory resources. □

As seen in the above simple example, when writing a script, one would usually want to be able to define and reuse transactions, requiring further (non-functional) constructs in the language. Similarly, one may want to add compile-time computations which calculate constants to be used in the rest of the script (e.g., memory usage of a class of actions could be automatically calculated as a function of CPU usage and duration). To avoid having to extend the language with such constructs, we have chosen to build the scripting language as an embedded language [6] in Python. This allowed us to avoid having to build a parser and type-checker for the language, and also allows the user to use Python for function definitions and computation.

4 An Application of SMock

SMock supports the generation of mock systems written in the Java language. As a case study SMock has been used to generate a mock document management system — an extension of the example given in Section 3.

In this case study, the generated system was used in conjunction with existing runtime verification tools in order to analyse the effects these tools have on the overall performance of concurrent systems, of which the document management system is an example. Two runtime verification tools, JavaMop [7] and polyLarva [2] were used in this case study. In both instances a sample property: *an edited document must always be saved before closing*, has been monitored.

The programs in Program 4.1 for JavaMop and polyLarva, show simplified excerpts of the specification scripts used to generate runtime monitors for JavaMop and polyLarva. The most relevant feature of these scripts is the definition of noteworthy system events, such as ① and ②, which one must necessarily hook onto, to monitor the required property. Our previous explanation of the SMock specification language has highlighted how these system events are created through the definition of named actions. Both RV tools use AspectJ[4] technology to convert the event specifications to aspect code that can identify the occurrence of these events on the system. The resultant code is woven into the mock system's code to provide runtime monitoring of its execution.

Program 4.1 JavaMop and polyLarva specifications

```
JavaMOP:
SavedDoc(Document d) {
① event open after(Document d):call(* Document.open(..)) && target(d) {}
   ...
   ere: (open save* edit edit* close)  }

polyLarva:
upon { newDocument(doc) } {
  events {
      ② open(doc) = {doc.open();}
     ...  }
  rules { ... }   }
```

The aim of this case study was not to carry out comparison between the runtime monitoring tools; but rather its purpose is that of highlighting the type of analysis that can be carried out using a mock system generated by SMock. In particular we wanted to note the effect of runtime monitoring on the system's performance when it is under considerable load. Changes to the specification affecting the choice settings result in different executions which model differing loads. We generated systems where only 10% of the users are carrying out document editing tasks and then increased this to 50% and 100% for the following executions. Since the property triggers only when a document is manipulated, this affects the overheads induced by the runtime monitors.

The analysis uses the profiling information from the runs of the generated document management system. For each configuration, the mock system was executed multiple times in the following setups: (*i*) without any code instrumentation; (*ii*) using JavaMop; and (*iii*) using polyLarva. Replaying was used to ensure comparison between identical executions.

The table below compares performance of the system execution and demonstrates how profiling data extracted by SMock can give a good indication of the effect that differing system loads can have on the overall system performance and execution time.

[4] `http://www.eclipse.org/aspectj/`

For each different load of users carrying out document management activities, the table shows the average memory usage and CPU usage across the whole execution together with an indication of system duration.

% users	**Average Mem**	**Average %**	**Total Sys Time (mins)**		
monitored	**(Kb)**	**CPU**	**None**	**MOP**	**poly**
10	13.8	31.3	1.6	1.7	1.7
50	24.3	32.9	3.2	3.3	3.3
100	27.2	33.5	3.4	3.5	5.7

The monitoring overheads are non-trivial in these scenarios and the analysis allows an understanding of how monitoring affects the overall system execution.

5 Related Work

While industry-calibre tools (e.g., jMock[5]) exist for mocking parts of a system under test, to the best of our knowledge, no tools exist which enable one to mock a whole system. Another significant difference of SMock from existing mocking tools is that these do not explicitly support the specification of CPU and memory usage. SMock, not only provides this through a dedicated language, but also provides constructs for the specification of probability distributions over such resources. These differences make SMock ideal to test systems which act upon other systems, e.g., monitoring systems and testing systems.

Another area of somewhat related work is the area of traffic generators (e.g., Apache JMeter[6]) for performance testing. However, such tools assume the existence of a system on which traffic is generated. Since the load on a runtime verification tool occurs *by proxy*, i.e. as a consequence of the load of another system, performance testing tools cannot be used directly for the performance testing of runtime verification tools.

6 Conclusions and Future Work

With the significant advancements in the area of runtime verification in recent years, the availability of mature tools is crucial for the increased adoption in industry. To facilitate the testing and profiling of runtime verification tools we have presented SMock[7], a mock system generator coupled with replay and profiling capabilities. The tool has been successfully applied to two state of the art tools showing the overheads in terms of time taken, memory consumption, and processing resources.

Future improvements to SMock will focus on providing more advanced profiling features such as power consumption measurement and automatic measurement repetition to ensure that results are not affected by external factors such as garbage collection

[5] `http://jmock.org/`

[6] `http://jmeter.apache.org/`

[7] The tool can be downloaded from `http://www.cs.um.edu.mt/svrg/Tools/SMock`.

or unrelated operating system activities. Furthermore, we would like to build an appropriate test suite to test runtime monitoring tools for correctness. Although, at the moment we only support the generation of a mock system written in Java, the design of the tool makes it straightforward to extend to other languages — which we plan to do in the near future.

References

1. Bodden, E., Hendren, L.J., Lam, P., Lhoták, O., Naeem, N.A.: Collaborative runtime verification with tracematches. J. Log. Comput. 20(3), 707–723 (2010)
2. Colombo, C., Francalanza, A., Mizzi, R., Pace, G.J.: Extensible technology-agnostic runtime verification. In: FESCA. EPTCS, vol. 108, pp. 1–15 (2013)
3. Colombo, C., Pace, G.J., Abela, P.: Compensation-aware runtime monitoring. In: RV. LNCS, vol. 6418, pp. 214–228 (2010)
4. Colombo, C., Pace, G.J., Schneider, G.: Dynamic event-based runtime monitoring of real-time and contextual properties. In: FMICS. LNCS, vol. 5596, pp. 135–149 (2008)
5. Colombo, C., Pace, G.J., Schneider, G.: Larva — safer monitoring of real-time java programs (tool paper). In: SEFM. pp. 33–37 (2009)
6. Hudak, P.: Building domain-specific embedded languages. ACM Computing Surveys 28(4es), 196 (1996)
7. Meredith, P.O., Jin, D., Griffith, D., Chen, F., Roşu, G.: An overview of the MOP runtime verification framework. JSTTT 14(3), 249–289 (2012)