Extracting Runtime Monitors from Tests

Christian Colombo Mark Micallef Keith Spiteri
PEST Research Lab
Department of Computer Science
University of Malta, Malta
{christian.colombo|mark.micallef|keith.spiteri}@um.edu.mt

The effort and training required to specify runtime monitors in industry might be one of the factors explaining the slow uptake of the technique. In this position paper, we discuss ways in which runtime monitors can be extracted automatically from tests which are typically available with industrial systems.

This brings with it a number of challenges, however, mainly because tests are typically focused on checking very specific behaviour, rendering the checks practically unusable in a runtime verification setting where the behaviour is user-directed rather than test-specified. In this context, we present an experiment and preliminary results which should provide the basis for further future exploration.

1 Introduction

A large portion of the software development industry relies on testing as the main technique for quality assurance while other techniques which can provide extra guarantees are largely ignored. A case in point is runtime verification [3, 12, 5, 13] which provides assurance that a system's behaviour is correct at runtime. Compared to testing, this technique has the advantage of checking the actual runs of a system rather than a number of representative testcases.

Based on experience with the local industry, one of the main reasons for the lack of uptake of runtime verification is the extra effort required to formally specify the correctness criteria to be checked at runtime — runtime verifiers are typically synthesised from formal specifications (e.g., Larva [5], JavaMOP [13]). One potential approach to counteract this issue would be to use the information available in tests to automatically obtain monitors [9]. The plausibility of this approach is the similarity between tests and runtime verifiers: tests drive the system under test and check that the outcome is correct; runtime verifiers also check that the outcome is correct but let the system users drive the system. Notwithstanding the similarities, the fact that in runtime verification the users drive the system means that, while tests are typically focused on checking very specific behaviour, runtime verification assertions need to be able to handle the general case. This makes it hard to create runtime monitors (which are generic enough to be useful) from tests, in particular due to the following aspects:

- **Input:** The checks of the test may only be applicable to the particular inputs specified in the test. Once the checks are applied in the context of other inputs they may no longer make sense. Conversely, the fewer assumptions on the input the assertion makes, the more useful the assertion would be for monitoring purposes.
- **Control flow:** The test assertions may be specific to the control flow as specified in the test's context with the particular ordering of the methods and the test setup immediately preceding it.
- **Data flow:** The test may also make assumptions on the data flow, particularly in the context of global variables and other shared data structures — meaning that when one asserts the contents of a variable in a different context, the assertion may no longer make sense.

Submitted to:
FESCA 2016

- **External state:** A similar issue arises when interacting with stateful external elements (e.g., stateful communication with a third party, a database, a file, etc.): if a test checks state-dependent assertions, the runtime context might be different from the assumed state in the test environment.

In view of these potential limitations, most tests do not provide useful information for monitoring purposes (due to their narrow applicability). Thus, one approach would be to build sifting mechanisms to select the useful tests while avoiding others which would add monitoring overheads without being applicable in general. Unfortunately, deciding which tests are general enough for monitoring is typically a hard if not impossible task to perform automatically since the assumptions the developer made when writing the assertion are not explicit. One solution is thus to explicitly ask the tester for the assumptions made. In practise this would however be impractical as the number of questions involved is significant: at least one question for each issue mentioned above.

Another approach would be to target tests which are less specific — *more abstract* — by design: While unit tests would typically be very low level and focused on a particular method, system level tests perform a number of steps which potentially touch upon several modules. Due to their higher level of abstraction, such tests make less implicit assumptions on the context in which they run. For example, Gherkin [10] tests are typically implemented for generic input with no assumptions on the order in which the tests are to be run.

In this position paper, we report on exploratory work which has been carried out both on converting unit tests (Section 3) as well as converting Gherkin tests into runtime monitors (Section 4). Next, we report preliminary empirical results (Section 5) and compare our approach to related work (Section 6), concluding with future avenues of exploration in the final section.

2 Background

This section touches upon unit and system-level testing, particularly the jUnit and Gherkin tools, as well as runtime verification, providing an overview to enable the reader to understand the work presented in the rest of the paper.

Unit Testing Unit testing involves having suites of numerous test cases, each targeting different parts of the system under test. It executes tests in an isolated manner with setup and teardown code invoked before and after each test. The underlying idea is to simply test the behaviour of a particular module and not the way it interacts with the rest of the system or its execution history. Therefore assertions implicitly assume that each test is run on a freshly set up system and takes into consideration the actual inputs used in the testcase. jUnit[1] is the de-facto unit testing API for Java which runs test cases from within the test environment.

Gherkin Gherkin [10] is a business readable, domain-specific language which lets users describe what a software system should do without specifying how it should do it. Gherkin tests take on a so-called *Given-When-Then* format as illustrated in Example 1 and the underlying concept is that a business user can specify what is expected from a system using a series of concrete scenarios. *Given* some precondition, *When* the user does something specific, *Then* some postcondition should hold true.

Example 1 *Consider the following Gherkin scenario:*

[1]http://junit.org

```
Scenario: Navigate to lab results page
Given I am on the doctors landing page
When I click on laboratory results
Then I should go to the lab results page
```

In this example, the client has specified one scenario as part of a feature on an online health system: if laboratory results are clicked, then user should be taken to the lab results page.

While useful as a specification language, testers use tools such as Cucumber [6] to specify browser interacting code (using tools such as Selenium[2]) for each step in the Gherkin scenario, called *step definition*, resulting in having automatically executable tests: Each test will execute a sequence of steps to make the (*Given*) precondition true, then carry out an action (*When*) and check that the post-condition holds (*Then*). For the example above, the Selenium script would include opening the doctors' landing page, clicking on the laboratory results, and then checks that the browser ends up in the lab results page.

Runtime Verification Runtime verification [3, 12, 5, 13] encompasses a set of techniques aimed at generating monitors which observe the system behaviour at runtime and check that it adheres to some specification.

To avoid the possibility of having erroneous monitors, these are typically generated automatically from formal specifications such as finite state machines. We note that such specifications are generic in the sense that one specification symbolically describes the expected behaviour of the system under all circumstances rather than for any ones in particular.

Considering the above, we note that runtime verification monitors are essentially test oracles which check user behaviour rather than specific testcases. This leads us to attempt to extract monitors from tests in the sequel.

3 Converting Unit Tests into Monitors

In our initial experiments with converting tests into monitors, we started out considering jUnit tests. This primarily consisted of parsing the tests to identify the sequence of method calls leading to the assertion, and then translating this sequence into a pattern detected through aspect-oriented programming [11]. Whenever the pattern is matched, the same assertions of the test are executed.

Example 2 *For example consider the following unit test which tests a withdraw method:*

```
@Test
public void withdrawTest(){
   double balance = 1000;
   double amount = 250;
   Account account = new Account();
   account.setBalance(balance);
   account.withdraw(amount);

   assert(getBalance() == balance - amount);
}
```

[2]http://www.seleniumhq.org

The resulting monitor would match sequences of setBalance *and* withdraw *method calls (and binding the parameters) and execute the assertion.*

We note that while in this case the assertion is valid for any values of balance and amount, in real-life execution traces, it is probably unlikely to find sequences where *withdraw* is preceded by *setBalance*. Furthermore, while the approach just described could generate generic assertions for the above example and similar ones where the assertion does not refer directly to literals, due to the implicit assumption within typical unit tests, we could not find a way of avoiding posing the following questions needed to testers:

Parameter generality: Is the parameter fixed for the assertion to hold? This question deals with whether the test assertion requires that the parameter used in the test is fixed. If answered in the affirmative, the monitor would be constrained to match (at runtime) the method calls leading up to the assertion only if the parameter is the same as the one used in the test.

Transition generality: Do the steps leading to the assertion need to happen strictly in the order as they appear in the test? When there are a number of method calls leading to the assertion, the sequence adopted in the test is sometimes crucial for the success of the assertion. In cases when this holds, the monitor has to ensure that the pattern matching requires the same method sequence.

State generality: Does the assertion only hold if starting from an initial system state? Unit tests generally assume a tear-down and setup after a previous test, giving the illusion that each test starts at the initial system state. If the assertion takes this into consideration, then the monitor should only perform the assertion if the pattern is matched starting from the initial system state.

Clearly, requiring testers to answer such questions would not make the approach too popular. Even worse is the fact that unit tests are typically very specific by design. Therefore one would expect that the answers to the questions above would typically be in the negative, meaning that the resulting monitors would not be very useful as they would rarely match with actual runtime executions.

In what follows, we consider higher level tests which are implicitly more generic.

4 Converting Gherkin Tests into Monitors

As in the case with unit tests, the first concern when extracting monitors from tests is to identify the pattern where the test assertion is applicable. In the case of unit tests, which are specified directly in Java, the options are limited. However, in the case of Gherkin, we have additional options at our disposal since the steps leading to the assertion are specified in terms of browser interactions. These interactions are then forwarded as network requests to the backend, where these requests are finally serviced in terms of a sequence of Java method calls. The advantage is that these three layers provide us with different options through which we can identify the pattern leading to the assertion:

Browser interactions At this level, monitors will be concerned with the user's direct interaction with the web application including user actions such as button clicks and text entry.

Network interactions Monitoring network interactions deals with network requests such as HTTP requests which are triggered as a result of user browser interactions.

Backend operations Finally, network requests trigger operations in the backend which deal with the requests.

We note that the different points of observation constitute different levels of abstraction since for example there might be various browser interactions which would trigger the same network request (such as a login which might be triggered by going through the homepage or a sidebar). Similarly, there might be different network requests which are handled by overlapping backend operations. (However, in this work we only focus on the client-side rather than the backend, leaving the latter for future work.)

Example 3 *Consider the following Gherkin scenario being executed and recorded:*

```
Scenario: Navigate to lab results page
Given I am on the "doctors landing page"
And I have pending lab results
When I click on laboratory results
Then I should go to the "lab results page"
And I should see my pending lab results
```

Recording the scenario at the browser level results in something like the following:

```
GIVEN step
- Observed navigation to: 'https://myhealth.gov.mt/doctors-landing-page'
AND step
- No recorded activity
WHEN step
- Observed click on: id='lab-results-button'
- Observed navigation to: 'https://myhealth.gov.mt/results-page'
THEN step
- No recorded activity
AND step
- No recorded activity
```

Converting the above observation into a monitor would result in listening for navigation to the doctors' landing page, checking that there are pending lab results using the code written in the step definition and subsequently, any click on the lab results should lead to the results page, once more executing the corresponding code in the step definition.

If instead of looking at browser interactions, we look at network requests, we get the following observations:

```
GIVEN step
- Observed request to: 'https://myhealth.gov.mt/doctors-landing-page'
AND step
- Observed request to: 'https://myhealth.gov.mt/search-lab-results?doctor=
"Borg"'
WHEN step
- Observed navigation to: 'https://myhealth.gov.mt/results-page'
THEN step
- No recorded activity
AND step
- No recorded activity
```

We note that now certain observations are missing — the button click in this example — but on the other hand other observations become visible, namely the search request to retrieve the lab results from the server. Therefore, the monitor which is synthesised in this case, first observes navigation to *the doctors' landing page. When this is followed by a search request, then this means that the monitor should check that the search results are next displayed on the following page.*

Discussion
We note that while in this example the resulting monitor was useful, in other cases where the assertion is more specific, the generated monitor might not be useful. For example consider the example:

Scenario:
```
Given I am on any page in the domain
When I click on the login button
And I provide correct login credentials
Then I should be logged in
And I should be able to see my username
```

If the checking of the login credentials in the test assertion is specific to the test case, then the test-to-monitor conversion cannot be fully automated.

5 Empirical Evaluation

To compare the two levels of abstraction in terms of the runtime overhead of the resulting monitors, we carried out an empirical investigation.

The overheads incurred as a consequence of using a listener alongside the SUT were measured. The measurements taken focused on two resources, memory usage and execution time latency. Memory usage was calculated as the average memory used over a fixed span of time. This measuring technique is considered to be typical in RV [8].

Table 1: Average Listener Memory Usage

Level	Memory (MB)
Browser - Selenium IDE	3.5
Network - Firebug	5.5

On the other hand, in order to measure latency, an automated setup had to be used such that dependencies on user input are eliminated (For example, time taken to hover mouse over button and clicking it). The automated Cucumber scenarios used in the initial stages of monitor generation were considered to be ideal, as they offer a fixed set of steps which can be easily timed. Furthermore, it was decided that the initial launch of both the local server and the respective plugin (Selenium IDE and Firebug) were not to be included in the resultant latency, as they can be considered as a one-time process, and hence do not hinder SUT performance. A set of three scenarios were used involving navigation to a different page, user log in, and adding an item to the shopping cart.

First, the scenarios were executed without launching the listener, thus producing a control execution time. The process was repeated, this time launching the listener at the start of each scenario. The latency is the difference in the time taken for the two different setups to finish execution, taking the average of three runs for every test case. The results show that the Browser level listener corresponded to higher latency. This can be attributed to the increased amount of captured events at this level of abstraction, thus requiring more messages to be sent over the web sockets.

When observing the resultant overheads, one notes that for most web applications these would not be detrimental to the overall system performance. Albeit reasonably low overheads, they may still be

Table 2: Execution Time Latency

Level	**Time Taken (s)**	**Time Taken w/ Listener (s)**	**Latency (s)**
Browser - Selenium IDE	44.335	45.241	0.906
Network - Firebug	44.297	44.928	0.631

problematic in cases of web applications which require very specific response time thresholds. In such cases, every millisecond might push the system over the specified limits, and thus the latency factor should be treated with great care.

6 Related Work

Testing and runtime verification are intimately linked — the main difference being that tests drive the system under test as well as give a verdict on the behaviour correctness, while runtime verification observes and checks user behaviour. As such, assertions in tests can be as specific as the test cases are, while in runtime monitor the assertion has to cater for all the possible user inputs. Therefore, in general it is not possible to generate runtime verification monitors from tests. However, a subset of testing which relies on a model to generate more varied test cases, known as model-based testing, is closer to runtime verification due to it being more abstract.

The idea of translating tests into test models is not new: Arts et al. [2] attempt to generate QuickCheck automata from EUnit tests, while in previous work [4], the authors show how to generate QuickCheck automata from Gherkin specifications. Furthermore, work also exists which translates test models into runtime verification: in particular, Pace and Falzon [9] show how QuickCheck automata can be converted into runtime verification monitors.

This work connects the dots and attempts to bridge the gap between Gherkin specifications and runtime monitors. However, while the end result sounds as if the two approaches might be piped together, the approach presented in this paper goes directly from Gherkin to monitors by recording the test during its execution.

7 Conclusions and Future Work

This paper presents and discusses the idea of generating monitors from tests. While the two are linked since both check runtime behaviour, the checks available in tests might be too specific for the particular test case. Following an initial exploration of generating monitors from unit tests, we turned our attention to system-level tests — Gherkin tests in particular — which are generally more abstract. To convert the test steps into the observable events, we recorded the test execution at different points: browser interactions and network requests, and reported on the empirical results of both approaches.

We conclude the paper by noting a number of limitations and discuss possible solutions for future exploration:

Privacy and encryption While recording browser interactions in a test environment, this might in practise not be possible to do when the system is used by a real system. A similar problem occurs when monitoring network requests, particularly so since requests would be encrypted. Our comments on this is that having monitors is not only useful for monitoring actual user traffic, but actually be more useful during the exploratory testing phase when usually the tests carried out are still mostly checked by hand.

Test specificity As noted earlier, it might be impossible to generate useful monitors from some tests simply because the logic in the test is too specific. There are a number of ways in which we are trying to address this.

Domain specific language One way of having better tests is to ask/help the tester to write more generic tests in the first place. Another way of looking at it would be to create help the testers create monitors first, i.e., the checking part of the test, and then create a test for the monitor using existing test generation techniques [1]. Later, upon deployment, the driving part of the test would be discarded and only the monitoring part would be kept. However, we do not wish to go down this route so that our approach is more easily taken up by industry.

Machine learning Another avenue we are exploring in this regard is to use existing language inference techniques so that we attempt to generalise assertions automatically. In particular, we are currently exploring tools such as Daikon [7] to extract invariants from test runs. The challenge would then be to filter out invariants which are not likely to be interesting, or filter out the results when such invariants are monitored.

We hope that through this work, we can bring about the advantages of monitoring systems without the disadvantages of having to learn and use technologies which are not currently mainstream.

References

[1] Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold & Phil Mcminn (2013): *An Orchestrated Survey of Methodologies for Automated Software Test Case Generation*. *J. Syst. Softw.* 86(8), pp. 1978–2001, doi:10.1016/j.jss.2013.02.061. Available at `http://dx.doi.org/10.1016/j.jss.2013.02.061`.

[2] Thomas Arts, Pablo Lamela Seijas & Simon Thompson (2011): *Extracting QuickCheck Specifications from EUnit Test Cases*. In: *Proceedings of the 10th ACM SIGPLAN Workshop on Erlang*, Erlang '11, ACM, pp. 62–71, doi:10.1145/2034654.2034666. Available at `http://doi.acm.org/10.1145/2034654.2034666`.

[3] Séverine Colin & Leonardo Mariani (2005): *Run-Time Verification*. In: *Model-Based Testing of Reactive Systems*, *LNCS* 3472, Springer, pp. 525–555.

[4] Christian Colombo, Mark Micallef & Mark Scerri (2014): *Verifying Web Applications: From Business Level Specifications to Automated Model-Based Testing*. In: *Proceedings Ninth Workshop on Model-Based Testing, MBT 2014, Grenoble, France, 6 April 2014.*, *EPTCS* 141, pp. 14–28, doi:10.4204/EPTCS.141.2. Available at `http://dx.doi.org/10.4204/EPTCS.141.2`.

[5] Christian Colombo, Gordon J. Pace & Gerardo Schneider (2009): *LARVA — Safer Monitoring of Real-Time Java Programs (Tool Paper)*. In: *Software Engineering and Formal Methods (SEFM)*, IEEE, pp. 33–37.

[6] Ian Dees, Matt Wynne & Aslak Hellesoy (2013): *Cucumber Recipes: Automate Anything with BDD Tools and Techniques*. Pragmatic Bookshelf.

[7] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz & Chen Xiao (2007): *The Daikon System for Dynamic Detection of Likely Invariants*. *Sci. Comput. Program.* 69(1-3), pp. 35–45, doi:10.1016/j.scico.2007.01.015. Available at `http://dx.doi.org/10.1016/j.scico.2007.01.015`.

[8] Ylìès Falcone, Klaus Havelung & Giles Reger (2013): *A Tutorial on Runtime Verification*. In Georg Kalus Manfred Broy, Doron Peled, editor: *Engineering Dependable Software Systems*, *NATO Science for Peace and Security Series - D: Information and Communication Security* 34, IOS Press, pp. 141–175. Available at `https://hal.inria.fr/hal-00853727`. Summer School Marktoberdorf 2012.

[9] Kevin Falzon & Gordon J. Pace (2013): *Combining Testing and Runtime Verification Techniques*. In: *Model-Based Methodologies for Pervasive and Embedded Software*, *Lecture Notes in Computer Science* 7706, Springer, pp. 38–57.

[10] Gherkin: *Gherkin Wiki*. Available at `http://github.com/cucumber/cucumber/wiki/Gherkin`.

[11] Gregor Kiczales (2005): *Aspect-oriented programming*. In: *Software Engineering (ICSE)*, ACM, p. 313.

[12] Martin Leucker & Christian Schallhart (2009): *A brief account of runtime verification*. *The Journal of Logic and Algebraic Programming* 78, pp. 293 – 303.

[13] Patrick O'Neil Meredith, Dongyun Jin, Dennis Griffith, Feng Chen & Grigore Roşu (2012): *An Overview of the MOP Runtime Verification Framework*. *International Journal on Software Techniques for Technology Transfer* 14, pp. 249–289.