Facilitating and extending software testing through monitoring techniques

white paper

Christian Colombo, Mark Micallef and Gordon Pace
Department of Computer Science
University of Malta
name.surname@um.edu.mt

While the quality of unit testing has improved dramatically over the past years, wider-ranging integration and system testing still proves to be challenging:

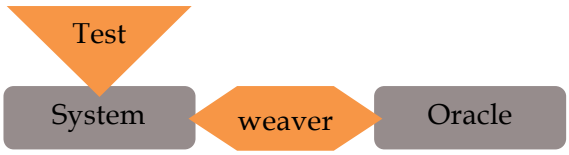
1. Complex behaviours which are difficult to programmatically check for
2. System-wide properties which result in cluttered code
3. Exorbitant number of test cases to cover most possible scenarios coupled with the safety-criticality of the software and the limited time to market

1. Checking for complex behaviours

A central challenge in software testing is the creation of the test oracle, i.e. the mechanism which decides whether a test passes or fails. Programming oracles in the same programming language as the system being tested is arguably as error prone as the programming of the system itself. A less risky solution is to express behaviours in terms of more abstract terms which are subsequently automatically compiled into code.

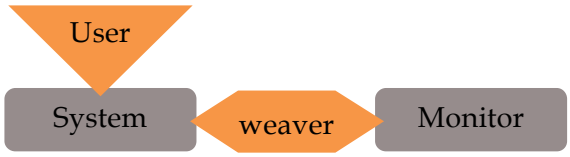
2. Inserting checking code without clutter

While programming the oracle might be hard, another problem is the integration of the oracle into the system under test so that it can make any necessary observations. Runtime monitoring provides the technology to alleviate this problem by injecting code automatically throughout a system (at runtime or compile-time) from within a single module. Thus, monitoring provides a clutter-free solution whereby oracles can be fully contained within a single module while still enjoying system-wide visibility.



3. Covering all system behaviours

Generating all potential test cases is virtually impossible for any system of reasonable size. Yet, if the oracle is kept in place beyond the testing phase as a monitor, then one would effectively be checking all (observed) behaviours for correctness. If a violation is detected at runtime, then the monitor can either simply flag the problem or can potentially also trigger error handling code.

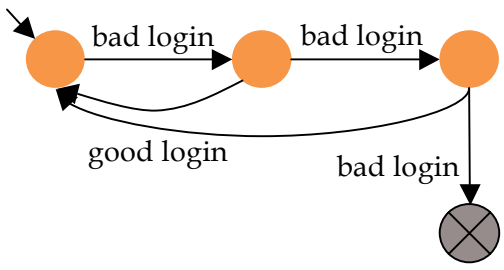


Tool support – Larva

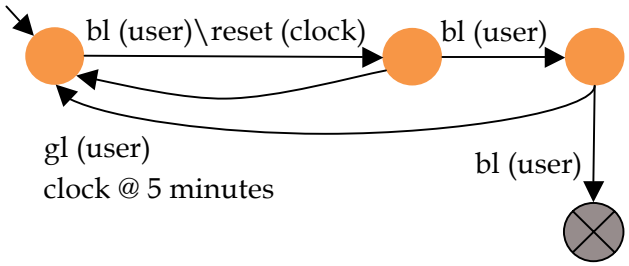
Larva is an easy-to-use tool which provides means for addressing the above-mentioned challenges. Specifically, Larva enables developers to perform:

1. Behaviour specification and compilation

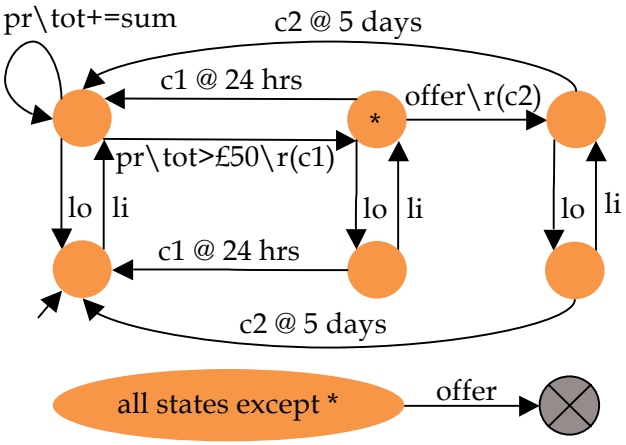
Larva enables the user to specify behaviours in terms of state machines which are in turn compiled directly into code. Consider the following example where we check that no more than two bad logins can occur consecutively:



While this example is admittedly trivial, one can already appreciate that understanding such a state machine is arguably easier than going through the code and understanding the updates of some form of counter throughout the login module. Moreover, Larva state machines also support advanced features such as clocks and parametrisation. Consider the following revised example (note the abbreviated event names):



Through the use of a clock which is reset upon the first bad login, we can now ascertain that the three consecutive bad logins have to occur within five minutes. Furthermore, by considering event parameters, we can distinguish between each user within the system and instantiate monitors accordingly. While the above examples were kept simple for illustration purposes, the described behaviours may be significantly more complex. For example consider a promotional offer which should only be given if a user has done purchases totalling more than £50 while logged in during the previous two days. The customer can only take up the offer during a 24 hour period and in such a case, no similar offer can be availed of in the next five days. This behaviour can be encoded as (li = login, lo = logout, r=reset, pr=pruchase):



Note that such a monitor is significantly more maintainable than the equivalent oracles written in code.

2. Attaching the monitor to the system

Larva enables the user to use system events such as method calls, exception throws, etc., as transition triggers for the state machines. For example, the following Larva script excerpt would associate any method call to any method named “loginAttempt” with an event called “login”:

```
login = { *.loginAttempt () }
```

Frequently, the user would also need to bind parameters such as binding the user object which is attempting the login:

```
login(User u) = {*.loginAttempt (u)}
```

Furthermore, distinguishing between a successful or a failed login might require the consideration of the return value as follows:

```
bl(User u, Boolean res) =
{*.loginAttempt(u)uponReturning(res)}
filter {res == false}
```

While monitoring can operate directly on the byte code, the script writer requires good knowledge of the source code and has to ensure that all relevant events are matching correctly.

3. Managing runtime monitors

The predominant issue in the context of runtime verification is the runtime performance of the monitor which might negatively affect the system’s performance or interfere with its normal operation. Depending on the seriousness of this concern, there are numerous options of how runtime monitors can be deployed:

During testing – Monitors are enabled during testing, particularly advanced testing stages such as alpha and beta testing and then switched off altogether upon deployment.

Offline – Monitors may be run on the system log on separate processing resources or when the system has excess resources (e.g., at night).

Online – Monitors are run on the live system, with hooks in the code which give monitors a direct communication link. The communication link might be blocking, i.e. the system waits for the monitor to process each event; or non-blocking, meaning that the system simply communicates events to the monitor and continues its execution. While this option is the most intrusive, it has the advantage of enabling the monitor to take corrective measures as soon as a problem is detected.

Sampled (on/offline) – Monitoring may be sampled over a selection of the system behaviour (e.g. only on a number of user sessions). Furthermore, sampling may occur either using some statistical distribution, or using other measures such as the trust attributed to a particular user account.

Case Studies

Over the years a significant number of case studies have been carried out which can be loosely classified under the following headings:

- **Security-critical systems**

Runtime verification, particularly online and blocking, is particularly useful in the context of security-critical systems where security is the main concern. In such extreme cases, designers would typically by ready to sacrifice performance in favour of full step-by-step verification. The most prominent work in this area is the application of runtime verification to the Mars Rover at NASA where mistakes are very costly.

- **Transaction systems**

Errors in transaction systems such as payment gateways are also taken very seriously since any defect might remain undetected while malicious users may be exploiting it in their favour (e.g. a few cents' mistake in the currency exchange mechanism under particular circumstances). Yet, in such cases designers value performance very highly too so sampled, offline or non-blocking runtime verification might be preferred. An example of work in this field has been carried out by the University of Malta in conjunction with Ixaris Ltd – a virtual credit card provider, processing thousands of transactions daily.

- **Highly-dynamic systems**

Runtime verification is also used in highly-dynamic environments such as web services where the runtime circumstances are difficult to predict and mock. Thus, various research groups such as the Polytechnic of Milan, Italy, have focused on the monitoring web service compositions, particularly a WS-BPEL named Tele-Assistance which enables a hospital to remotely monitor the condition of patients after being discharged.

Summary and Conclusion

Faced with modern challenges of ensuring the correctness of computer systems which are growing ever more complex, runtime monitoring provides a number of promising solutions:

1. Aids the specification of complex behaviours by providing compilation from an abstract language aimed for use by software developers.
2. Enables the process of weaving-in the code which checks for the correct behaviour at runtime.
3. Supports various flavours of runtime monitoring architectures, designed for different use cases.

The technique has already been used in a number of industrial case studies but we aim to encourage wider adoption by providing support, knowledge and tools – hopefully alleviating some of the effort currently put into the creation of highly dependable systems.

Further reading

1. Gordon Pace, **Trusting a Computer with your Money**, Economic Update, January 2011.
2. Christian Colombo, **FSM-based Monitoring and Runtime Verification**, Interview on InfoQ, September 2011.