Extracting Runtime Monitors from Tests: What we have learned so far

Luke Chircop*, Christian Colombo, Mark Micallef, Adrian Francalanza, and Gordon J. Pace

University of Malta - Computer Science Department

Runtime verification is a technique that can be used to provide extra guarantees that the system being monitored behaves correctly at runtime. Unfortunately, this technique is rarely adopted in the industry due to various reasons. One of which revolves around the extra resources and effort required to integrate the technology.

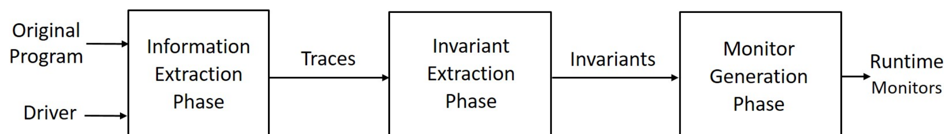

Fig. 1. Three phase approach to automatically generate runtime monitors

A three phased process shown in Figure 1 was therefore designed for the automatic generation of runtime monitors, consisting of: the extraction of behavioural knowledge, the inference of invariants, and finally the translation of invariants into runtime monitors. The abstract is going to be discussing the challenges and decisions that had to be taken.

Extracting Knowledge

Using dynamic analysis, invariants can be inferred just by observing how the system behaves and reacts to a set of given inputs. The amount of behaviour observed will directly affect the quality of the invariants that are produced. It is therefore crucial, that the chosen tests exercise as much of the system's functionality as possible using a variety of inputs. Knowledge is then obtained by recording the system's state before and after the execution of every event in the form of traces.

A variety of tools including Daikon [2], BCT [3] and DySy [1] exist, that are able to record the state of primitive type objects (like Integers, booleans, etc.) appertaining to argument values, global variables and return values. To the best of our knowledge no tool exists that is able to record all types of objects with their correct representation, as a consequence, this may introduce inaccuracies when inferring invariants. Whilst evaluating whether such tools could be used in the industry, another limitation was encountered involving the size of traces generated. These became overwhelmingly large impeding the invariant inferring process from completing successfully.

Inferring Invariants

Once knowledge about how the system behaves is gathered, this can be processed to infer a number of invariants based on either control or data flow. For the purpose of our research, we focused on inferring data flow invariants modelled as data invariants. Furthermore, we are using a readily available tool called BCT [3] which internally uses Daikon [2] to infer these data invariants.

* Project GOMTA financed by the Malta Council for Science & Technology through the National Research & Innovation Programme 2013

```
1 transactionsystem.UserAccount.withdraw(double):::ENTER
2 this.balance > amount
3 transactionsystem.UserAccount.withdraw(double):::EXIT
4 amount one of { 100.0, 500.0, 1000.0 }
5 this.balance < orig(this.balance)
6 this.balance > orig(amount)
7 this.balance - orig(this.balance) + orig(amount) == 0
```

Listing 1.1. Examples of invariants that can be produced

Listing 1.1 shows examples of invariants that can be generated for a *withdraw* function inside a banking system. Useful invariants such as requiring that the *balance* is greater than the *amount* to be withdrawn can be observed in line 2. Other valuable invariants can be noted between lines five to seven. However, we have noticed that there will be some invariants that might still be too specific, catering for only a subset of the possibilities. An example can be seen in line 4, where an invariant was inferred checking that the *amount* value is either 100, 500 or 1000. Clearly, this does not reflect all the possible inputs that the *withdraw* function can accept. Therefore, a filtering mechanism is required to be able to identify and prune such invariants reducing the number of false negatives. This is currently not implemented and is part of the future work.

Translating Invariants into Runtime Monitors

Having extracted the beliefs, these can then be translated into runtime monitors and integrated with the live system to provide free extra guarantees. Although this process is quite straightforward, a number of decisions have to be taken. The first of which, is whether the generated monitors are going to observe and verify the behaviour of the system synchronously or asynchronously. This may be crucial for parts of a system that are safety critical and cannot afford to have additional overheads introduced at runtime. Another decision that needs to be taken revolves around how the monitor is going to react once a violation is identified. Currently, we suggest and are only reporting the observed violations. However, we are foreseeing the possibility of also blocking the action which causes the violation.

Conclusion

Runtime verification is seen as an approach that could help the software industry to obtain extra guarantees that the software behaves correctly. To make the approach easier and more appealing to use, a process has been designed whereby unit, system, integration and other similar tests are translated into runtime monitors automatically using a dynamic analysis approach. Design choices had to be taken with regards to; how and what information should be extracted, the techniques used to infer invariants, and how the invariants are to be translated into runtime monitor. Finally, the need for a mechanism that prunes the invariants inferred was also identified and discussed as potential future work.

References

1. C. Csallner, N. Tillmann, and Y. Smaragdakis. Dysy: Dynamic symbolic execution for invariant inference. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 281–290, New York, NY, USA, 2008. ACM.
2. M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3):35–45, Dec. 2007.
3. L. Mariani and M. Pezz. Behavior capture and test: Automated analysis of component integration. In *proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems*, 2005.