# Improving the Automatic Runtime Monitor Generation Process via Pruning: A way forward

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

University of Malta - Department of Computer Science

Software quality and robustness are elements that the industry is constantly striving to achieve. However, no guarantees can be given that the end product is bug free. Our research proposes an automated process whereby, existing tests are translated into runtime monitors. This is achieved through a three step process of extracting behavioural knowledge, inferring conditions (modelled as invariants), and finally translating them into runtime monitors. Multiple existing tools such as Daikon [1] have primarily been designed for testing purposes to extract invariants. As a consequence, some invariants inferred have been observed to be unsuitable for the purpose of runtime monitoring; potentially generating false negatives.

Determining whether an inferred invariant is valid for runtime monitoring can prove to be a highly challenging and a non-intuitive decision to make. Invariants that are valid can be described as conditions that accurately model how the system state or behaviour should be or behave at particular relevant program points. Listing 1.1, Line 9 illustrates a valid invariant checking that the balance after a withdraw request has been registered correctly and is able to do so for all possible inputs that the system accepts. However, inferred invariants may over or under approximate the behaviour that it is trying to verify.

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

**Listing 1.1.** Examples of invariants that can be produced

As [3] states, an over approximating invariant, is one that accepts more behaviour than that accepted by the real system. On the other hand, under approximating invariants may exclude behaviours that can occur in the real system. Examples of under approximating invariants are illustrated in Listing 1.1, lines 4 and 6. Line 4 checks that the country is equal to Malta upon a withdraw request, which is a clear under approximation since a bank typically allows withdrawals from most, if not all, countries. Line 6 shows an invariant that under approximates the possible *amount* inputs for a withdraw request to 100, 500 and 1000 euros, which is not correct since any value greater than 0 should be accepted.

It is near impossible to avoid having over or under approximating invariants. Therefore, a number of techniques are currently being explored to automatically detect and prune said invariants.

**Introducing Ideas for Pruning**

An approach that has been considered, takes advantage of the developers experience and introduces an iterative process of refinement.

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

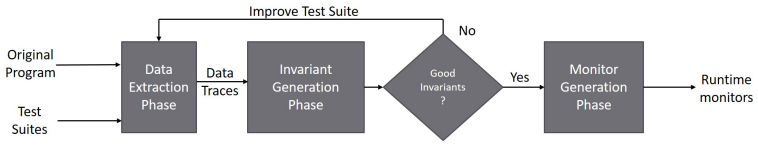


**Fig. 1.** Iterative approach to prune unwanted invariants

As illustrated in Figure 1, tests are used to exercise the system under test to infer invariants. Before the invariants are translated into runtime monitors — particularly suspicious invariants such as the "one of" kind — a developer determines whether they are correct or require further refinement. For example, the invariant illustrated in line 6 would be identified as under approximating the behaviour suggesting that not enough varied inputs where used in the tests. Thus, the developer can then decide to either modify or introduce additional tests with different values or behaviour; producing a more refined and valid invariant such as "amount >= 0".

Mutation testing techniques are also being considered to evaluate whether the inferred invariants are valid.

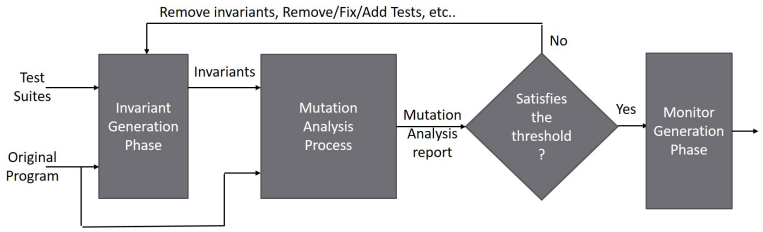


**Fig. 2.** Three phase approach to automatically generate runtime monitors

This technique involves modifying small bits of the system under test, for example replacing the subtraction inside a withdraw function with an addition and exercising the tests to identify which invariants flag the incorrect behaviour as a violation. Consequently, the remaining invariants that did not flag a violation are considered to be unuseful. Developers or automated techniques such as Genetic Algorithms [2] can then use the acquired list of valid and invalid invariants to determine whether further refinement is required to fine tune the invariant inference process, as shown in Figure 2.

Other approaches such as the combination of data and control flow invariants are also being considered, aiming to prune as much as possible unwanted or incorrect invariants that introduce false negatives.

## References

1. 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.
2. S. Ratcliff, D. R. White, and J. A. Clark. Searching for invariants using genetic programming and mutation testing. In *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*, GECCO '11, pages 1907–1914, New York, NY, USA, 2011. ACM.
3. P. Tonella, A. Marchetto, C. D. Nguyen, Y. Jia, K. Lakhotia, and M. Harman. Finding the optimal balance between over and under approximation of models inferred from execution logs. In *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, ICST '12, pages 21–30, Washington, DC, USA, 2012. IEEE Computer Society.