Lifelong Verification of Software Systems

Christian Colombo Mark Micallef Gordon J. Pace
Department of Computer Science
University of Malta
Email: christian.colombo | mark.micallef | gordon.pace@um.edu.mt

***Abstract*—Computers systems are increasingly interacting with our day-to-day life, but for this interaction to be facilitating and supporting, rather than interfering with our actions, these systems have to be dependable and trustworthy. The area of system verification and validation has a long history in computer science, but scaling up existing approaches to complex and large real-life systems is still an open-ended research question. In this paper we summarise and relate several ongoing research projects and tool development efforts in this field taking place within the Department of Computer Science.**

I. Introduction

The degree of impact of computer system failures has been monotonically increasing since the advent of computers, mainly due to the increased human interaction with, dependence on and the delegation of tasks to computers. In order to address this challenge, much research has gone into techniques to improve the development process, from software engineering looking into ways to organise and manage better the development cycle in order to decrease the incidence of bugs, to formal methods looking into mathematical techniques to have sub-systems developed correct by construction or allow them to be verified *a posteriori*. Despite these efforts, the increase in size, complexity, decreasing time-to-market and prevalence of systems still result in the incidence of errors increasingly offsetting the improvement brought about by the software-improvement techniques.

One important realisation is that bugs and failures are inevitable, and they have to be addressed as a part of the system lifetime. Current industrial practice usually sees the treatment of dependability and reliability in two ways: (i) a quality assurance team specifically focussing on identifying problems in the software and which is active throughout the system specification and development phase mostly using testing techniques; and (ii) the development team, which incorporates monitors, usually in the form of assertions and checks, for invariants and system properties directly or indirectly in the code (despite the fact that the logic in the code should safeguard from such assertions and checks to ever fire). If we were to simplify the system lifetime as shown in Figure 1 into development and production phases, it results that the quality assurance team's main activity is in the first phase, in which they try to identify as many problems as possible, while the defensive work of the development team plays a primary role post deployment, in the production phase.

The verification problem in the two phases poses different challenges, and it is thus not surprising that the methodologies of the two approaches are distinct. The verification process during the development phase has to deal with the fact that (i) the specification is still being sketched out, (ii) parts of the system have not yet been developed, (iii) bugs are still abundant in the system, and (iv) user behaviour must be emulated in order to discover bugs. On the other hand, post-deployment, the main challenge is that monitoring induces an overhead on the system which is usually undesirable.

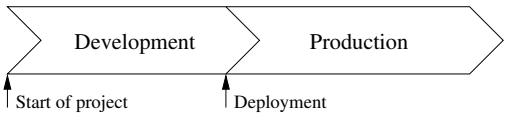

Fig. 1. Simplified timeline of a software project.

In this paper we start by outlining testing and runtime monitoring, the two primary technologies used in the two distinct phases (see Sections II). We then proceed to propose a framework combining the two approaches (Section III), reducing duplication of effort which takes place when the two approaches are addressed separately. We emphasise how our proposed approach deals with the distinct verification needs of the different phases. In particular, we argue for the use of domain-specific languages for writing the specification, which enables us to use a single specification artefact for verification throughout the system's lifetime. The work builds upon recent research taking place within the Department of Computer Science focussing on testing and runtime verification.

II. Verification of Systems

The development of software systems can be loosely split into two phases (shown in Figure 1): the development phase and the production phase. Due to their distinct characteristics, each of these phases poses particular challenges and possibilities from a verification point of view. In what follows, we attempt to articulate the significant differences in these phases and how these affect the verification process.

Development Phase
During the development stage the main involvement of the user is for the purposes of refining the specifications of the system. The specifications are typically in flux at this stage and evolve regularly. In this context, initially, verification at this stage typically deals with low-level unit specifications which are tested using unit tests. Due to their low level, unit tests do not typically involve the user.

As the development progresses, testing becomes higher level and system level testing starts to take place. During this phase, the system is typically still too unstable for it to be tested directly by the user. For this reason, user behaviour is modelled into tests so that in the absence of an actual user testing the system, the modelled user behaviour (shown in Figure 2 as a user in a dotted box) is used to drive the system.

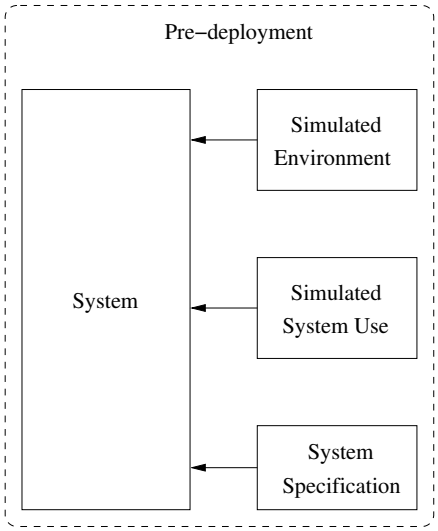

Fig. 2. During testing, the user interaction with the system is emulated.

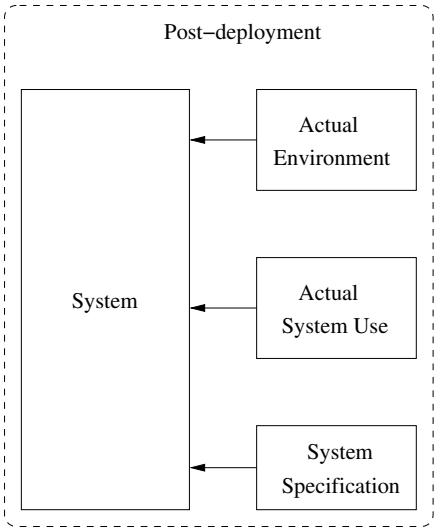

Fig. 3. During runtime verification, the actual user's interaction with the system is available.

The main overall aim of verification in the development phase is that of detecting as many bugs as early as possible. Thus, the emphasis is not on the efficiency of the verification code, or the guarantees it can give, but rather on how effectively can the verification technique detect a large number of bugs at the early stages of the development phase.

Production Phase
Once the developed system is completed up to some user-accepted criteria and gets deployed, the concerns of the verification effort changes — efficiency becomes of utmost importance as verification during the runtime of a deployed systems eats up resources which would otherwise be used to serve user requests, and it becomes crucial to detect *any* bad behaviour. The main reason for this contrast is that the purpose of verification in the development stage is to aid developers detect bugs while the aim of verification during production is to ensure the user gets to experience the expected system's behaviour. Consequently, the stickman illustrated in Figure 3 represents the actual user as opposed to a modelled user. In fact, an actual system usage pattern might be significantly (or completely) different to the modelled user behaviour. The reasons may include lack of knowledge about typical users of the system, insufficient time for testing a representative set of behaviours, etc.

In the subsections below, we elaborate more on the two verification techniques of testing and runtime verification. In particular, we will be expanding on the characterising features and how these are being supported through various projects.

A. *Testing*

Testing activities aim to build confidence in a system by uncovering as many bugs as early possible. The predominant mechanism for doing so is that of exercising the system under scrutiny in a variety of ways such that a sufficiently large coverage of potential user behaviours is achieved. Confidence is built by monitoring the discovery of new defects until such a time when it is determined that the discovery of new severe defects is sufficiently unlikely. Of course, a failure to find defects in a system does not guarantee that such defects do not exist. This lack of guarantees renders testing a risk-based activity whereby engineers attempt to exercise systems in ways that (in their assessment) are most likely to expose issues.

In comparison to a decade ago, the industry has achieved significant progress where testing is concerned, even though approaches tend to vary significantly between organisations [1][2]. However, there is a drive to incorporate testing early in the development process and share its responsibility across different roles (developers, testers and customers). There is a widespread belief that tools and techniques such as test driven development [3], continuous integration [4] and automated system-level regression tests [5] empower developers to confidently adapt the system to fluctuating specifications in the knowledge that there is a safety net of tests to sound the alarm if anything is broken. Despite this, problems in production still persist, mainly due to the reasons discussed below.

Incomplete and dynamic specifications
In a dynamic market prone to disruptive innovations, customers commission software development in the believe that it will help them gain a competitive edge [6]. Consequently, their vision of what the software should do changes continuously in response to demands from the market in which they operate. This results in specifications which can be incomplete, vague and highly prone to change. Needless to say, building and testing software when the goalposts are continuously moving is a challenging task with the final specifications arguable only being known when the decision is made to deploy a system to production.

Incomplete System
Although development teams have access to environments in which to deploy and test a system, it is rarely the case that such environments are complete. For example, if a system interacts with a number of third party services, such services would not be present in a pre-production test environment. This leads to extensive use of mock components during system testing which, while helpful in simulating specific scenarios, arguably reduce the realism of the environment in which the system is tested.

Limited number of real users
During system development, testing largely depends on exercising the system in a way in which it is expected that real users will interact with it once deployed to production. The problem with this approach is that it is very difficult to predict all realistic interactions which the variety of individuals who will use the system will have with it. That is to say that whilst engineers aim to achieve as much coverage as possible, there will likely always be a subset of users whose particular behaviours are not explicitly tested. If the system exhibits undesirable behaviour in an untested user interaction, the inevitable result is the propagation of that behaviour to a production environment.

It is worth noting that during testing, overheads introduced by test drivers are not of particular concern. In fact it is normal practice for large (possibly inefficient) test suites to be executed overnight with developers finding test results available in the morning. This enables them to enter a systematic test-fix-repeat cycle throughout system development with the aim of continually reducing risk and building confidence. In this context, testing can be seen as a first line of defence which eliminates a large percentage of problems prior to system deployment albeit without providing significant guarantees. This can then be reinforced by *runtime verification*, a second line of defence that is capable of providing guarantees in a production environment. This is the subject of the following section.

B. Runtime Verification

Currently in the industry, runtime monitoring is largely limited to inlined assertions and consistency checks as part of the code of the system. However, many tools [7], [8], [9] are now appearing, which allow the separation of the specification from the code, and which automatically weave in the properties into the code.

Runtime verification is typically applied once the system is deployed, resolving a number of problems and challenges arising from the testing process, but posing its own particular challenges. By being applied directly on the actual runtime behaviour generated from the system's interaction with real users, runtime verification does not have the problem of having to mock the context of the system's environment or user behaviour. Also, since it is applied post-development, all the system units are available, circumventing the problem of having to mock units which may not yet have been fully developed. On the other hand, the fact that monitoring occurs at runtime introduces a new challenge — the monitor has an impact on the system's performance, which should not interfere with its functionality. We discuss these issues in more detail in this section.

Resource constraints
A major concern of runtime verification architectures is the usage of runtime resources whose consumption might affect the user's experience. There are several techniques to mitigate such a problem:

Offline monitoring
To provide continuous guarantee that the observed system behaviour adheres to the specification, runtime verification is run in sync with the executing system, and thus potentially slowing it down. To avoid consuming precious runtime resources, runtime verification can however be applied on the runtime trace but out of sync with the system execution, freeing the system from having to wait for the verifier's guarantee. Naturally this is less safe but any bugs occurring at runtime are still caught, albeit with a lag. This technique has been applied successfully on an industrial case study [**?**] dealing with financial transactions. In the case of financial transactions, it is particularly crucial not to cause any delays for the user experience — especially during a surge in usage which is a typically occurring pattern in such systems.

On/offline monitoring
The downside of opting for a purely offline runtime verification approach is that by the time the violation is detected, the system would have progressed further, making it difficult to take any reparatory actions. One way of compromising between online and offline monitoring is to enable the monitor to switch across the two modes at runtime. Switching from online to offline is easy as one would simply need to stop enforcing the synchronisation between the system and the monitor. On the other hand, to revert back to synchrony, there are two main options [10]: if the synchronisation needs to happen before the monitor has detected any problems, then the system can be paused while the monitor catches up with the system by processing all the pending events. If the monitor has detected a bug during asynchrony, then the monitor cannot "catch up" with the system since the monitor cannot progress upon violations. The alternative is to somewhat undo the behaviour of the system following the violation using some sort of compensation mechanism [11] — similar to what is typically used in processing long-lived transactions.

Actual user behaviour
A significant consequence of applying verification post deployment is that the verification process has access to the actual system behaviour. This means that guaranteeing that the system behaviour is correct only requires one to ensure that the current system behaviour is correct. This contrasts sharply with the highly challenging guarantees required during development which have to take into consideration any possible system behaviour. In fact, while ensuring that a system works correctly under any circumstance is generally intractable for non-trivial systems, ensuring that a single system behaviour is correct is tractable for a wide range of correctness properties.

Fixed specifications
Another advantage of operating in the post-deployment environment is that by that stage, the system's specifications would have stabilised (as opposed to highly fluctuating specifications during the development stage). Such an environment makes it worthwhile to invest in specification analysis, particularly to contribute towards the optimisation of their verification. One such technique involves static analysis [12] to identify parts of the specification which do not need to be verified at runtime because it can be ascertained a priori that the system can never violate such a specification. Static analysis attempts to relate the behaviours of the system vis-a-vis the behaviours prohibited by (parts of) the specification; if the two sets of behaviours do not intersect, then the specification or part thereof contributing to those behaviours can be ignored during runtime verification.

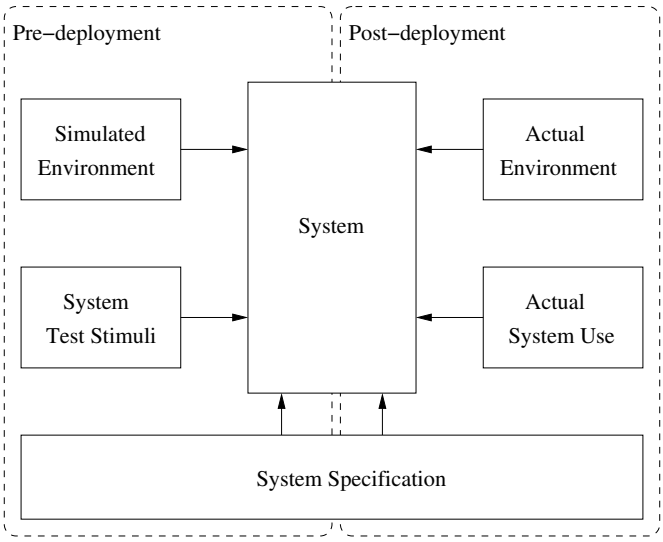

Fig. 4. Framework to combine testing and runtime monitoring

III. Combining Verification Techniques

Clearly, the two verification approaches of testing and runtime monitoring act orthogonally in terms of when they are used in the lifetime of the system. However, the scope of the verification coverage remains largely unchanged — both testing and runtime monitoring would, for instance, be interested to verify that across a whole transaction, the sum of monetary transfers across accounts adds up to zero. Through the separation of specification and implementation, and automated techniques to transform the specification into testing oracles and monitors, it would be possible to use a common artefact for testing and monitoring.

Although one can approach the challenge by investigating how artefacts produced for monitoring can be used to test a system, this approach is not practicable due to (i) the chronological order in which the two phases take place; and (ii) the fact that few software development companies take a structured approach to monitoring and largely use *ad hoc* assertions and runtime checks.

Given that testing is practically universally adopted and strictly regulated within software development companies, migrating the testing oracles to be useful for monitoring ensures that the approach is immediately applicable in practice. The main challenge is that testing requires two elements — the oracle which assesses whether a particular behaviour is correct, and the test-case generator, which emulates the user and environment behaviour, and which in most cases are given intertwined. Furthermore, in approaches such as mock testing this information may be combined further with information about how to deal with parts of the system which have either not yet been developed or which one may desire to emulate (for efficiency or other reasons) during testing. Decoupling these elements is not always straightforward.

Our proposed framework in which these components are combined together can be found in Figure 4. We have already investigated different frameworks to migrate testing information into runtime monitors. In [13], we have looked at the transformation of specifications written in QuickCheck [14] (a commercial testing tool targeting systems written in Erlang) into monitors written in Larva [7]. The test specifications in QuickCheck combine trace generation with correctness conditions, but through careful decomposition of the description it was possible to generate effective runtime monitors. Similar to the approach with QuickCheck, we are currently investigating ways in which monitors can be extracted from JUnit tests. This is somewhat more challenging than working with QuickCheck since JUnit tests are typically not as generic as QuickCheck tests, thus making it more difficult to extract monitors which are applicable to observed runtime behaviour.

Other than this work, there is not much other prior research in this direction. One notable exception is [15] which enables the developer to express complex assertions in terms of monitors instead of basic assertions. This is very useful when attempting to verify assertions which span over multiple snapshots of the system's state. Examples include checking the sequence of method calls, checking time delays between particular events, etc.

We are currently looking into more effective and complete ways in which to go from test specifications to monitoring ones. One challenge we still have to address is that of reducing monitoring overheads. Another is that tests are sometimes expressed for very specific cases, based on the knowledge of the test engineer that that value generalises to a large class of inputs. For instance, the test engineer may test a financial transaction system on transactions of $1 with the insight that if the system works for this value, then it will work for all values between the $0.01 to $999.99 range. We believe that such choices should be documented as part of the test suite — information which can be used to generalise the monitors to check the results for all values, rather than just for that of $1.

From key issue which arises in this context is the language used to express the specification. Specification languages need to be fit for purpose, not only from a correctness perspective, but also from the point of view of being useful in a modern software engineering context. Customers typically specify systems in ambiguous natural language and are thus likely to continuously refine (if not drastically change) these specifications. This, combined with the current approach of systematically interpreting natural language specifications in order to develop a system and subsequently utilising yet another language to define properties for monitoring purposes provides ample opportunity for error, not to mention acting as a substantial barrier to entry for monitoring techniques. We argue that domain specific languages (DSLs) can play a key role in this regard. Such languages can provide enhanced expressiveness within the context of a particular domain and can be ubiquitously used for specification, testing and monitoring purposes. Furthermore, they can be used as a common language by both technical and non-technical stakeholders thus reducing the potential of miscommunication. We argue that whilst formal languages and logics utilised thus far do have a role to play in lifelong verification of software systems, this role should take the form of being the solid-yet-invisible foundation on which domain specific languages are designed. In doing so, stakeholders can communicate in a language with which they are intimately familiar whilst behind-the-scenes tools can extract information about expected system behaviour from test scripts written in DSLs and leverage it to

construct useful monitors for added assurance in a system's post-deployment lifetime.

IV. Conclusions

With ever increasing pressures to release new software versions, fast changing requirements and higher expectations of software reliability, software development companies face a continual balancing act: the quality of the software and the time in which it reaches the market. To date most of the effort of software quality assurance occurs in the pre-deployment phase through testing. While highly effective, testing cannot guarantee the absence of bugs and furthermore the quantity and realism of the test cases are limited due to time pressures and lack of tests involving real users. Another approach to software assurance is runtime verification, a technique which unlike testing is applied post-deployment, can detect any bugs occurring at runtime while an actual user is using the system. Applicable to different phases of a software system, these techniques are highly complementary, enabling quick detection of bugs in the pre-deployment stage, and guaranteed bug detection upon occurrence in the post-deployment stage.

A number of approaches which aim at combining testing and runtime verification exist and are also being developed with the aim of exploiting the advantages and avoiding duplication of effort. In the future, we aim to continue exploring synergies between the two techniques while attempting to answer practical questions which would necessarily arise when the two techniques are adopted in industry. For example, given that in runtime verification the oracle executes with the deployed system, should it be the responsibility of the developers to write, or should it be considered as part of the testing regime which is never switched off? The answer is not straightforward and we look forward to investigate such questions through industrial case studies.

References

[1] A. Bertolino, "Software testing research: Achievements, challenges, dreams," in *2007 Future of Software Engineering*. IEEE Computer Society, 2007, pp. 85–103.

[2] E. Engström and P. Runeson, "A qualitative survey of regression testing practices," in *Product-Focused Software Process Improvement*. Springer, 2010, pp. 3–16.

[3] L. Madeyski, *Test-Driven Development*. Springer, 2010.

[4] S. Stolberg, "Enabling agile testing through continuous integration," in *Agile Conference, 2009. AGILE'09.* IEEE, 2009, pp. 369–374.

[5] M. Fewster and D. Graham, *Software test automation: effective use of test execution tools*. ACM Press/Addison-Wesley Publishing Co., 1999.

[6] L. Cao and B. Ramesh, "Agile requirements engineering practices: An empirical study," *Software, IEEE*, vol. 25, no. 1, pp. 60–67, 2008.

[7] C. Colombo, G. J. Pace, and G. Schneider, "Larva — safer monitoring of real-time java programs (tool paper)," in *SEFM*, 2009, pp. 33–37.

[8] M. Kim, M. Viswanathan, S. Kannan, I. Lee, and O. Sokolsky, "Java-mac: A run-time assurance approach for java programs," *Formal Methods in System Design*, vol. 24, pp. 129–155, 2004.

[9] P. O. Meredith, D. Jin, D. Griffith, F. Chen, and G. Roşu, "An overview of the MOP runtime verification framework," *JSTTT*, 2011, to appear.

[10] C. Colombo and G. J. Pace, "Fast-forward runtime monitoring - an industrial case study," in *Runtime Verification, Third International Conference, RV 2012*, ser. Lecture Notes in Computer Science, vol. 7687. Springer, 2012, pp. 214–228.

[11] C. Colombo, G. J. Pace, and P. Abela, "Compensation-aware runtime monitoring," in *RV*, ser. LNCS, vol. 6418, 2010, pp. 214–228.

[12] W. Ahrendt, G. J. Pace, and G. Schneider, "A unified approach for static and runtime verification: Framework and applications," in *International Symposium On Leveraging Applications of Formal Methods Verification and Validation (ISOLA'12)*, 2012.

[13] K. Falzon and G. J. Pace, "Combining testing and runtime verification techniques," in *International Workshop on Model-based Methodologies for Pervasive and Embedded Software (MOMPES'12)*, ser. LNCS, 2012.

[14] J. Hughes, "Quickcheck testing for fun and profit," in *Practical Aspects of Declarative Languages*, ser. LNCS, 2007, vol. 4354, pp. 1–32.

[15] N. Decker, M. Leucker, and D. Thoma, "junit$^{\text{rv}}$-adding runtime verification to junit," in *NASA Formal Methods*, ser. Lecture Notes in Computer Science, vol. 7871. Springer, 2013, pp. 459–464.