Industrial Experiences with Runtime Verification of Financial Transaction Systems: Lessons Learnt and Standing Challenges⋆ ⋆⋆

Christian Colombo[1] and Gordon J. Pace[1]

Department of Computer Science, University of Malta
{christian.colombo,gordon.pace}@um.edu.mt

Abstract. The chapter will focus on experiences the authors had in applying runtime verification in industrial settings, in particular on financial transaction systems. We discuss how runtime verification can be introduced in the software development lifecycle and who are the people to be involved and when. Furthermore, we investigate what kind of properties have been found useful in practise and how these were monitored to keep intrusion to a minimum. Next, we describe two significant case studies which have been successfully carried out in the past, and conclude by outlining a number of challenges which we believe still need to be addressed for runtime verification to become more mainstream in industrial settings.

1 Introduction

As software systems increased in size and complexity, it was quickly recognised that many problems which arise in system development can be addressed by adopting a well-defined, more rigorous process, moving from an individual-based craft view of programming to a process (and team) based engineering approach [29]. Different software engineering processes have been advocated and adopted by industry, and today it is unthinkable that any non-trivial software be developed in an ad-hoc manner. These software engineering processes have shaped the organisation of industry, and any novel element part of the software development process stands little chance of being adopted in the short-term unless it finds a home as part of this organisational structure. Runtime monitoring and verification have been advocated as very industry-friendly techniques, especially due to their scalability to large systems, and accessibility to traditionally trained software engineers. Surprisingly, however, the literature describing the use of runtime verification in industry and evidence of its adoption remains sparse and far between. Use of formal methods (and in particular runtime verification) in the literature tends to consider the adoption of formal tools throughout the development process — for instance, in [28] one of the few papers which reports on the experience of integrating

⋆ The Open Payments Ecosystem has received funding from the European Union's Horizon 2020 research and innovation programme under grant number 666363.

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

runtime verification into the development process of an industry-grade project, model checking is initially used to verify at the design and code level, thus providing formal properties to be used with the runtime verification tool, an unrealistic assumption in most industrial settings.

One can argue that dynamic monitoring and verification has featured in software development since the first software systems — adding auxiliary code to check what the system is doing, and assertions to check predicates in order to identify and report or react to unexpected behaviour, is nothing but a primitive form of runtime monitoring and verification. More recent structured approaches to runtime verification, which focus on separating the concerns of system development and the specification of monitors and verification code, allow for greater independence between the process of system and monitor development. However, the integration of runtime verification in standard software engineering practice remains a little explored area. Although some work (e.g. [37]) does look at how the development process can be adapted to incorporate runtime analysis concerns, a shift in existing software engineering practice is difficult to achieve, and thus, integration into existing practice is crucial to widespread industrial adoption of runtime verification.

In the past decades, as software dependability became increasingly important, testing was promoted to a first-class concern in the development process, with approaches such as test-driven development becoming the norm in many settings. It is natural to ask whether runtime verification can simply piggy-back onto the integration of testing in the software engineering process. The most important common concern between testing and runtime verification is the development of oracles able to flag unexpected system behaviour. However, the two also differ substantially in other aspects, making their merging in the software engineering process difficult. The fact that, unlike testing, runtime verification code (sometimes) is intended to be executed alongside the system post-deployment puts extra demands on this code, and requires a spread in the concern of software engineering from mainly the development time towards the runtime [6]. Also, although oracles and verification checks have a similar goal, in practice tests tend to hard code input behaviour and output pairs, whereas in runtime verification it is necessary to abstract the oracle to all potential input behaviour. These differences indicate that depending on the quality assurance infrastructure already existent in most software companies, adopting runtime verification might not be as straightforward as one might hope.

In this chapter, we present a number of industrial case studies which we have been involved in, and discuss what worked and what issues arose in the process. A secondary aim of the chapter is to assess, albeit in a qualitative and anecdotal manner, the challenges runtime verification faces before it can be adopted in the industry. Furthermore, it is worth noting that all case studies discussed in this chapter are in the financial software sector.

Case studies 1 and 2 were carried out with two different companies[1]. Both cases were the outcome of employees from the companies attending research talks and showing interest in adopting aspects of the runtime technologies we spoke about. We then

[1] Due to non-disclosure agreements, one of the companies cannot be named. However, it is worth noting that both companies had a R&D team of 50–100 persons.

set up a process of giving hands-on talks on site to company employees about the more pragmatic aspects of runtime verification. This was followed by being on their site to work on an initial proof-of-concept implementation with the hope of bootstrapping the use of runtime verification in a more widespread fashion within the companies' products.

Despite the limited technological success of these initial experiences, the collaboration has led to two formal projects being setup with one of the companies, indicating that the technology does hold promise to the industry. *Project 1*[2] [13] is an ongoing project GOMTA (Generation Online Monitors from Tests) between the University of Malta and Ixaris Ltd. in which the focus was to address one of the challenges identified in our initial collaboration — that of specifying appropriate properties for runtime verification. *Project 2*[3] [5, 4] is another ongoing project with Ixaris Ltd, where the runtime verification aspect is more ambitious, since runtime verification is integrated as a core part of the compliance engine of the OPE (Open Payments Ecosystem) platform. Also of interest is that the development of the platform and the compliance engine are being done concurrently, unlike the other use cases, in which integration of runtime verification was attempted *a posteriori* to the system development.

All these use cases are discussed in more detail in the rest of the chapter.

2 Financial Transaction Systems

Over the past years, we have worked on various industrial financial transaction systems. In this section we combine the common aspects into a single description, highlighting any differences only when necessary. Although financial transaction systems face various challenges, from fraud and security, to functional correctness, all the work we discuss focusses on the functional correctness, since various third-party tools already address issues such as fraud detection and security effectively.

The transaction systems we interacted with, handle credit card transactions and are thus composed of two sub-systems: one which handles the part of the transaction taking place between the customer and the transaction system, and another handling the transaction between the transaction system and the bank. These will be referred to as the transaction handling system and the processor communication system respectively.

A transaction is processed by going through a number of states such as authorisation, communication with the user interface, inserting the transaction in the database and communicating with the commercial entity involved in the transaction. Each type of transaction will have its particular chain of states through which it must go to be successfully completed. Similarly, system user accounts also go through a cycle of events including registration, logging in, performing financial activities, suspension, etc. The interaction between these two life cycles as well as its implications on the amount limits are among the most commonly specified properties:

Life cycles Entities in a transaction system, particularly users and transactions, go through a life cycle of stages. Each stage will determine how the entity can behave and stages it can transition to in the future. For example, *a user who has been*

[2] https://www.um.edu.mt/ict/cs/research/projects/gomta
[3] https://www.openpaymentsecosystem.eu

suspended should not be allowed to perform any financial transactions. Similarly, *a transaction which is in the processing stage, should not be modifiable.*

Real-time A transaction system typically has real-time aspects such as *a transaction should not take longer than 500 milliseconds to complete*. Naturally, real-time properties can also be related to life cycles, e.g., *once a user has been inactive for three months, then the account should be frozen.*

Amount and frequency limits Other commonly occurring properties in financial transaction systems involve amounts and frequencies of transactions and transaction amounts, e.g., *a user cannot transfer more than €2000 a week*. These limits may also be related to the life cycle, e.g., *a user who has registered but has not yet been fully approved, cannot withdraw more than €100 per week from the account*.

Other There are a number of other properties which are difficult to classify under the previous headings. For example, to ensure adherence to VISA regulations, the transaction system cannot store credit card numbers. Another practical property is to ensure that a transaction is not initiated twice by the user mistakenly clicking the submit button twice.

3 Runtime Verification from a Process and Software Engineering Point of View

While making runtime verification attractive to industry necessarily requires the investigation of the appropriate process and software engineering practices, these elements remain largely unexplored in the literature.

The software testing community has had to solve a similar problem when it came to integrating testing in the development life cycle of software, requiring years of experimentation with different setups. To some extent engineering of properties for monitoring is similar to the engineering of test oracles in that both are meant to tag behaviour as good or bad by observing the system's behaviour. However, there is a major difference between the two, namely that test oracles are typically designed to handle only the test case it has been written for. On the other hand, a monitor oracle needs to be generic enough to handle any observed behaviour. This makes the problem significantly different and given the lack of published material on this topic in the context of runtime verification, in this section we simply give some anecdotal reporting on what we have done and how it worked out.

3.1 Process Engineering Challenges

The first problem when introducing a new verification technique such as runtime verification within a software company is to identify the people who will be interacting with the technique. This section will analyse a number of questions which arise from process engineering point of view.

Engineering the properties When attempting to start the process of engineering properties for our industrial partners, a number of questions started to emerge:

► **Which properties are useful to monitor** Identifying which properties are to be monitored might in itself be a challenging aspect for the success of a runtime verification project. If the involved people do not see the benefit of monitoring, then it is likely that monitoring will be sidelined. Our case studies have taught us that a number of meetings might be required before the right kind of properties are identified for monitoring (Experience 3.1). Even the notion of a property itself is usually alien in the context of the software development industry (typically the word *property* is taken to refer to an object attribute). However, once the initial communication hurdles are overcome, appropriate system-wide properties — typically having a temporal aspect — start to emerge.

Which properties are worth monitoring?
A number of discussions were needed with our industrial partners simply to identify which properties are worth the effort of monitoring. At first the example properties which were being suggested were deemed to be superfluous given the way the system had been engineered. For example checking that the balance is correct after a transaction was well tested, would have simply wasted resources to monitor it. Similarly, properties found in runtime verification literature at the time — mainly focusing on properties extracted from the Javadoc of Java libraries such as those concerning iterators, maps, etc. [10, 9] — while useful, were not deemed to warrant the introduction of runtime verification technology (opting instead to use code reviews, etc to eliminate such standard bugs).
Following more discussions and involvement of different people in the organisation, it started to emerge that the most useful properties were those which crosscut the system across its modules or history. The crosscutting nature of properties such as: *"ensure the credit card numbers are never stored inside our system"*, *"ensure a user does not carry out any transaction when suspended"*, or *"the user should follow a particular cycle throughout its lifetime"*, made it hard to check them (without monitoring) in a straightforward manner, i.e., without cluttering the code and risking introducing additional bugs in the process.

Experience 3.1. Taken from case studies #1,#2.

► **Who is responsible?** Once a number of example properties are identified, the next challenge is to identify who would be responsible to express them in a monitorable format. As Experience 3.2 shows, when the runtime verification engineer started to work at the site of our industrial partner, it was not straightforward to pinpoint the team which could most naturally handle property writing. The issues involved were not limited to who has the knowledge of the system at the right level of abstraction, but also who is willing to do the work while finding it beneficial (Experience 3.3). The conclusion of our case studies was that the people who tick all the identified boxes in a software development organisation are the QA personnel who have a

vested interest in ensuring that the system as a whole works as expected. Runtime verification provides them with a methodological approach of specifying properties and a way of automatically checking them.

- ► **In what format should properties be expressed?** Identifying an appropriate format for expressing the properties is crucial to enable the identified personnel to express the properties. Admittedly, we have not experimented with different specification languages. However, our use of automata-flavoured notation (more specifically [25]) has proved effective with non-academics who used it.

Which team will host the runtime verification engineer?
One of the interesting characteristics which emerged from both case study #1 and #2 was how many times the researcher had to change the team he was working with: In the case of the first case study, the researcher was first placed with the security team. Soon it was realised that the runtime checking of functional aspects had little to do with security. Next, the researcher was placed within the development team: This move facilitated the familiarisation of the researcher with the system code, but did little to help him understand the properties of interest. Next, the researcher had a meeting with the system architects and this proved to be a swift way of obtaining a bird's eye view of the system, including some of its main properties. Finally, the researcher found it best to work closest to the testing team whose system-level tests were closest to what the runtime monitors were expected to do. A similar experience of moving from one team to another could be recounted for the second case study. The situation was however different in that testing was mostly carried out by the developers themselves. This meant that developers were mainly responsible for testing their own modules while there was a dedicated team for quality assurance (QA) which performed some testing and dealt with customer issues. This time the researcher found it best to interact with the QA team to identify the properties of interest. The main difference between the kind of properties identified by the QA team and those identified by interacting with the architects is that the former are more likely to actually be violated at runtime (e.g., a fee which is charged twice to the customer), while the others are more fundamental but usually highly unlikely to be violated (e.g., the sequence of states a customer goes through: from registered, to active, to suspended, etc).

Experience 3.2. Taken from case studies #1,#2.

Engineering the verification code Once the properties are available, the next challenge is to engineer the corresponding code to check for their violation. Typically, the purpose of writing the properties formally is to exploit some runtime verification tool which is able to generate the code for the properties automatically. On the other hand, programming the verification code from scratch is also an option, but this would mean

Who will write the properties?
In both case study #1 and #2, when developers were asked to write properties to be monitored at runtime, they felt that they were simply redoing work already done while at the same time their view of the system was focused on their particular module, making it difficult to capture system-wide properties. Instead, runtime verification monitors were more naturally expressed by high level testers/QA personnel who view properties as a concise way of expressing complex system properties and providing them with a kind of dashboard through property violation reporting. Furthermore, since the people writing the properties were not the same ones who programmed the system, this approach yielded better results in identifying bugs.

Experience 3.3. Taken from case studies #1,#2.

that the property engineering step is skipped. Furthermore, taking this option would also usually mean that the code would not be separate from the system code, and consequently, this is programmed directly by the programmers (in the case of our first experience (see Experience 3.4) since the testing team was closely involved in the development process and had ample experience in writing system-wide scripts, it was responsible for integrating the runtime verification code). If the verification code is not integrated with the system — usually when the verification does not take place in sync with the system — then the code can be maintained by a team other than that of the system developers, e.g., the QA team (this was the case with the second experience mentioned in Experience 3.4).

Who manages the verification code?
In the first experience, since runtime verification was carried out in an online fashion, the testing team had to be involved to help set up the necessary scripts to integrate the monitoring within the system code. The reason for involving the testing team was that they had ample experience with launching the system through script writing. Unfortunately, the system was never updated after the introduction of runtime verification code, meaning that we cannot comment further on this experience regarding the management of synthesis and synthesised code.
In the second experience, runtime verification was carried out in an offline fashion and therefore this could be fully managed by the QA team with little involvement from the development team in case required logs were missing or in an unexpected format.

Experience 3.4. Taken from case studies #1,#2.

Considering the option of automated synthesis, two separate choices have to be made: (i) concerning the synthesis code and (ii) concerning the synthesised code. If the latter is to be integrated with the system, then one would typically expect the system developers to be responsible for it. However, if the generated code is to be used separately from the system, then once more there is the option of involving other teams. As for the synthesis code, since this does not directly interact with the system and it would probably be a third party tool, then its management need not be tied to the system developers' team.

Recommended procedure for introducing runtime verification Based on our experience, in an ideal scenario, we recommend the following procedure when introducing runtime verification in a company not familiar with the technology:

Initial meetings An initial meeting where an overview of the system and teams involved (including architects, developers, and QA) is provided by the company. Next, another meeting where the ideas behind runtime verification are presented to the teams by the runtime verification engineers.

Information gathering Following the meetings, all relevant specifications, architectural designs, etc. should be made available to the runtime verification engineers. This is then followed up with meeting with relevant parties to fill in any gaps in such documents.

Meeting with QA The QA team can provide information regarding the kind of problems they worry about the most on a day-to-day basis. These are usually the areas where runtime verification can be useful. Properties can then be composed based on these revelations.

Implementation phase Once properties are at hand, input is likely to be required the system architects and developers.

Testing phase Finally, when monitors are running, one would likely need to verify any detected violations with the QA team. It is probable that the first issues encountered would be the result of miscommunicated requirements, requiring fine-tuning of the properties.

Delegation phase If monitors are running as expected, then it would be the right time for the responsibility to pass on from the runtime verification engineers to the teams in their respective roles: the architects to ensure the monitoring code is well integrated in the system design, the developers to manage the code, and QA personnel to maintain the properties.

3.2 Monitor Design Challenges

A significant challenge from a software engineering point of view with respect to runtime verification is to keep the concern separate from the system's logic while at the same time making it easy to integrate the two.

Furthermore, this has to be achieved while keeping the runtime overhead to a minimum. The following subsections deals with elements one should be aware of when designing the verification code. It is worth noting that many of these elements are interconnected and one choice influences others. One important starting factor when considering these options is that of what properties one is interested in, and when and how one

is to react to their violation. Such considerations already restricts architecture choices, and event extraction mechanisms. The

Architecture Design One important question to be addressed is that of how the high-level architecture combining the system and the verification units is designed. In both case studies 1 and 2, the verification modules were developed *a posteriori*, and had to be integrated to systems which had been in production for various years, which proved to be an extra challenge in that limited architectural choices were available.

- ► **Synchronous vs. asynchronous vs. offline monitoring** One major choice to make when integrating runtime verification and the system-under-scrutiny, is whether the composition of the two is (i) *online synchronous*, in that after each relevant step, the system will pause for the verification component to complete and announce compliance before proceeding further, or (ii) *online asynchronous*, in which the monitor is running with the system but steps of the system are checked asynchronously i.e., the system continues as the monitor does its verification; or (iii) *offline*, in which the system simply dumps relevant information during its execution, and the verification is carried out completely independently of the system, possibly even after the system has finished executing. The choice of architecture impacts how much the monitors can help the system react to errors, but also the overheads of the deployed monitoring. We had different experiences with possible composition approaches, as discussed in Experience 3.5.
- ► **Managing communication between system and monitors** When monitoring takes place in a white box fashion, i.e., with full knowledge and access of the system code, monitors might be inlined directly along with the system code. This is typically done through the use of aspect-oriented programming [32] although it is common to write assertions by hand. To keep concerns more separate, e.g., if the system and the monitor are running on different resources and/or implemented in different technologies, one might opt for a less tightly coupled form of communication such as the use of TCP/IP [20]. When monitoring in a black box fashion, the separation between system and monitors is naturally bigger and thus less direct ways of communication would be typically employed. For example, the monitor might use a tracing facility at the virtual machine level to pick up events of interest. Similarly, the monitor might be able to indirectly detect system API calls by tapping into the system's communication channel. Opting for even less interaction, the monitor might simply process logs which the system would have saved in a database or text file during its execution.

Event Extraction Design Runtime monitoring requires an awareness of the system behaviour, typically by capturing relevant events[4]. In what follows, we describe three kinds of software events and outline ways these can be captured and communicated to the monitor:

[4] Other than software events, one may for example capture the state of the hardware, or perform regular sampling of the variables. However, in this chapter we focus on the more commonly used software events

How to synchronise between the system and the monitors?
In case study 1, we started by implementing online synchronous verification on a sandboxed system. However, in the second case study we had to forgo synchrony due to (i) lack of trust impeding the integration of the runtime verification tool as part of the development toolset; (ii) fear of overheads due to online monitoring impacting the system, particularly at times of peak transaction traffic. The solution initially adopted to enable verification was, in both use cases, to adopt an offline policy [1, 25, 26]. Given that the interested events of the systems were already logged by systems in use by both companies, it was simply a matter of accessing existing logs and connecting them with our runtime verification tool.[a] The results were sufficiently convincing that the monitoring was considered to be adopted on a nightly basis, running it on the logs of the day. This led to the realisation that an important feature of a runtime verifier is efficient bootstrapping — starting up the verification process in a fast manner, without having to rerun full historic traces every time. This led to a solution which was effective enough to be used in the nightly verification process [22].
The use of offline monitoring also enabled further trust in the verification package, which enabled further investigation, even if online overheads were still considered prohibitive since they were not planned for in the original system design. Financial systems typically handle *long-lived transactions* — financial transactions which last far too long to justify locking of resources (e.g., user's bank account) in order to ensure consistency. The solution practically universally adopted in this industry is that of using *compensations*, effectively computations which can approximate the undoing of part of a transaction. In this manner, transactions are allowed to proceed unchecked, and in case of a late discovered failure, the transaction is "rewound" to just before the event that broke consistency constraints. This led to the development of a novel quasi-synchronous runtime verification [24, 23] in which the monitor was deployed asynchronously (though online) with the system, but upon identifying a violation, compensations were triggered to enable recovery in the state of the system where the violation actually occurred.

[a] It is worth noting that although the required events were logged, many events unnecessary for our properties were also logged, so using the logs as a starting point for identifying points-of-interest in the system is not necessarily a useful procedure.

Experience 3.5. Taken from use cases #1,#2.

- ► **Method-call-based events** Method calls frequently provide the right correspondence between the system's behaviour and the monitor events of interest. For example if the monitor is interested in money transfers, probably one can easily find a method which performs the money transfer, providing access to the parameter representing the amount being transferred. Method entry and exit points are typically captured through aspect-oriented programming (this was the case with our case studies, see Experience 3.6), or a tracing mechanism which the virtual machine provides.
- ► **Communication events** While in Java it feels natural to capture method call entry or exit, other programming languages or system organisations may provide different useful points of interest. A prevalent one of these is message communication in the case of languages such as Erlang [19] or organisations such as the service-oriented architecture [18]. Once again, such communication can be captured using similar techniques, as applicable, such as aspect-oriented programming and tracing.
- ► **Events-by-design** Rather than relying on naturally occurring execution points in the system (such as method call entry/exit points and communication events), another option is to explicitly plan points-of-interest when to raise an event in the system design. From a monitoring point of view, this approach naturally represents the most straightforward one as the system emits events automatically without the need to capture them. At their most basic, such events may take the form of logging events in a text file or database. In other cases, events might be broadcast to interested subscribers, one of which might be the monitor.

How to capture system events?
In both case studies, the events of interest could be directly mapped to method calls. For this reason, it was natural to opt for method-call-based events. Furthermore, given the maturity of tools supporting aspect-orientation, we chose a well-known aspect-oriented extension for Java, AspectJ. A significant difference between the two case studies is that the first was carried out online while the second was carried out offline by connecting to a database. We note that AspectJ could not be used to directly interact with an SQL database. However, by using a Java event replayer we were able to use AspectJ for both case studies.

Experience 3.6. Taken from use cases #1,#2.

Verification Design Challenges Having events of interest reaching the monitor, we now focus on how the monitor will process them. The main concern in this respect is how to keep the runtime overheads to a minimum and avoid memory leaks which might cause the monitor to take more resources to the detriment of the system.

- ► **Keeping runtime work to a minimum** One choice when designing the verifier is whether to explore the monitored logic a priori to avoid having to unfold it during

runtime. For example in the case of LTL [35], one would generate the equivalent automaton such that at runtime one would simply need to move from one state to another rather than rewriting the formula. The approach we took in our case studies (see Experience 3.7) is to some extent even more extreme as we chose to allow the users to program the properties directly as automata[5]. In this way, we pass on the control of (most of) the overheads to the user.

► **Bounded resources and garbage collection** If the chosen specification language supports monitoring using bounded memory, one may carefully implement the verifier such that the resource boundedness is exploited. The approach adopted for the case studies was to have a fixed set of user-defined states and thus memory leaks can only be introduced by the user through the Java code which can be used in transition conditions and actions.
Furthermore, another concern is the garbage collection of monitors — unused monitors can cause a memory leak. In general it is not trivial to identify monitors which can be discarded, since monitors are typically stateful and discarding part of the state might lead to incorrect monitoring. For our case studies (Experience 3.7) we chose to allow the user to explicitly mark states as *accepting*, meaning that once an automaton reaches that state it can be garbage collected.

How to design the verifier?
For both case studies we used the runtime verification tool Larva [25] to generate the monitors. Two important choices in the generated verification code were: (i) to use explicit automata, meaning that at runtime only simple if-conditions are evaluated (apart from conditions and actions explicitly programmed by the user); and (ii) to generate a hashing function for monitors (building on the user-defined hashing function of the monitored object) so that monitor lookup takes place in constant time. The first case study, in particular, served as the first testbed for the Larva tool and several modifications were introduced based on the experience. One such modification is the introduction of *accepting states*, i.e., states which signify property satisfaction and hence that that particular automaton can be garbage collected. Providing a means of garbage collection proved crucial to have monitors which are usable in real-life.

Experience 3.7. Taken from use cases #1,#2.

3.3 Conclusions

In this section we have presented the main challenges we have encountered when introducing runtime verification into an environmental setting. The challenges fall under

[5] Users all had an undergraduate degree which covered automata and they did not have full formal training in using formal logics such as LTL, they were comfortable using automata.

two clear categories (i) how the introduction of monitoring will impact the management of the software design and development process, and (ii) the technical challenges as to how to capture events and process them, i.e. the monitoring architecture, for the system at hand. A number of observations we made from our experiences were the following:

1. Companies do not trust new software easily, especially if it interacts with their live system at runtime.
2. Overheads are a worry, even when they might not be a real concern.
3. A major challenge is to have existing company structures organised to fit their current software engineering process absorb runtime verification without reorganisation.
4. Attractive, low-cost applications of monitoring have been found to be statistics gathering and user interface traversal analysis.

4 Challenges in Adoption of Runtime Verification

After presenting the challenges and design issues involved in introducing runtime monitoring in industry, this section presents a number of proposals and describes how these are being taken on board in two ongoing projects:

Project 1 — OPE The Open Payments Ecosystem (OPE) is an EU-funded Horizon 2020 project, aiming at creating a single pan-European cloud-based marketplace allowing third party developers to create payments applications and service providers (e.g., banks) to provide a range of services (e.g., card authorisation, ACH transfer, Swift) to support these applications. As a core component of the OPE infrastructure, is a verification engine which allows for matching applications with service providers based on their requirements, and to runtime verify the behaviour of these applications to ensure compliance to legislation, risk restrictions and other rules as required.

Project 2 — GOMTA The GOMTA project — Generating Online Monitors from Tests Automatically — is a project funded by the Malta Council for Science and Technology (MCST). The project aims to facilitate the adoption of runtime monitoring by saving the user the specification of the properties, extracting them instead from the test suite.

4.1 Challenge 1: Monitoring Overhead

Based on our experience with industrial case studies, monitoring overheads (primarily time, but also memory) have proved to be a major challenge and hurdle in the adoption of runtime verification in industrial-grade systems. The runtime verification community has focussed on the use of techniques at two different levels of abstraction: system level monitoring vs. business logic. The former, focussing on elements of lower-level code and libraries (e.g., iterators), implies higher requirement of low overheads of monitoring due to the denser spread of events, while the latter can make do with higher overhead per event since the events being monitored are typically substantially sparser.

Industry tends to invest substantial resources in identifying the right infrastructure and libraries, with trustworthiness being one of the important metrics used. Due to this, it was observed that the use of runtime verification techniques was seen by the industrial collaborators solely as a means of verifying their business logic. This reduces the requirements as to what are reasonable overheads, but it is worth noting different issues related to overheads which have been identified in the past use cases:

Worst-case overheads: The main concern with overheads is how large they can grow per event. However, given that runtime verification is a technique which (may) use the history of the system to deduce correctness, a concern is also that certain properties might require more time to check as the history grows longer, unless techniques such as incrementally verifiable properties are used to ensure this does not happen.

Variability of overheads: Another concern is that the overheads might change as the system evolves, leading to variability in quality-of-service measures over time.

Overhead spikes: In many transaction systems, there are (sometimes predictable) spikes of usage. For instance, on the payment portal of an online betting service, one gets high numbers of transactions just before an important sports event. This results in a proportionate spike in overheads, but is also the moment when fast reactivity is of high importance. A decrease in transaction processing speed could have a proportionately decrease in income. Catering for these moments of high server stress through hardware redundancy is only part of the solution here, and techniques to deal with monitoring in the presence of such spikes is a challenge still to be addressed.

Throughput: In a financial transaction system, all the concerns above are ultimately transaction, rather than event centric. In other words, transaction throughput is a key measure used by this industry. This means than looking at overheads at the quantum of transactions (which are variable compounds of events) gives a better hold on the applicability of the techniques in this domain.

Many techniques have been developed in the runtime community to address the issue of overheads. From the adoption of additional hardware for verification e.g., using GPUs [8, 34] or FPGAs [31] to adaptive techniques to manage monitors through measures of criticality e.g., [7], much runtime verification literature is concerned with this issue. From a more pragmatic perspective, it is still the case that choosing which architecture to adopt — in particular whether online or offline monitoring — is largely motivated by the requirements on overheads.

Work on the use of static analysis techniques in order to partially verify requirements and thus alleviating runtime verification overhead is also showing promising results e.g., [16, 11, 30, 36, 3, 15, 38]. Recently, we have started adopting such a technique (in ongoing project 1 — see Experience 4.1), which uses static analysis to reduce dynamic properties, thus lowering overheads.

Combining static and dynamic analysis
In the OPE project, one important functionality of the framework is to enable a developer to submit a payment app (or rather a model of the app), which is automatically matched with an appropriate service provider, based on capabilities, risk analysis and other aspects. In order to perform this matching, static analysis of the model submitted by the developer is performed [4].
This gave the opportunity to include further static analysis to reduce runtime verification overheads in the compliance engine, which has to check that (i) the application adheres to the model supplied by the user; and (ii) that it does not violate legislation, service provider risk limits, etc. For example, according to English legislation, the customer should always have the possibility of redeeming money from his or her account after closure. Using the app model, we statically check that this possibility is in fact supported — noting that this would otherwise have to be runtime checked frequently (even when no redemption is carried out). Moreover, regulations also state that money redemption should occur at par value and without delay. However, it is not possible to statically verify that these hold as the model does not contain this level of detail, which leaves parts to be checked dynamically (in this case, for example, timely redemption is not statically verifiable at the level of abstraction of the model). These remaining checks are delegated to be carried out through runtime monitoring.

Experience 4.1. Taken from ongoing project #1.

4.2 Challenge 2: Proposals for Runtime Verification from the Software Engineering Point of View

Introducing runtime verification within a software development life cycle presents a number of challenges as highlighted in the previous section. In what follows, we attempt to address them below by describing different approaches we have adopted in ongoing projects.

Monitoring as part of system design One of the main drawbacks of our previous experiences was that monitoring was not included in the original design of the system being monitored. Instead, monitoring had to be somehow retrofitted into the system architecture. In the OPE project (see Experience 4.2) runtime verification was included from the start and used to ensure the reliability of the framework.

Monitor architecture The underlying system architecture naturally has a direct effect on the monitor architecture. In traditional monolithic systems, without significant effort, the choice is usually limited to online or offline monitoring. System architectures which allow submodules to be more decoupled such as actor systems and those based on the service-oriented architecture, allow more monitoring options. The OPE (see Experience 4.3) is based on a micro-services architecture and therefore it was natural to have monitoring as a service and the system may decide to wait or not for the monitor verdict depending on the context.

Extracting events Identifying system execution points of interest and intercepting them through aspect-oriented programming proved to be a non-trivial task in previous case studies. Having predefined, clearly specified events makes it significantly more straightforward for components within the system to communicate as the events serve as a common interface; not least for the monitor. In the OPE project (see Experience 4.3), events from each micro-service are published with the monitor simply listening out for the relevant ones.

Monitoring as part of the system design
In past case studies the monitor has always been introduced after the system had already been developed. On the contrary, in the OPE the compliance unit (of which runtime verification plays a major role) was part of the initial design of the framework. This saved the OPE execution environment from having to be inundated with checks to cater for the legislation. The design, in turn, was taken into consideration when choosing the implementation framework and as further elaborated in Experience 4.3, incorporating the monitor in the OPE was straightforward.

Experience 4.2. Taken from ongoing project #1.

Monitoring architecture in a micro-services architecture
Being programmed as a monolithic Java system, previous case studies relied on aspect-oriented programming to embed the runtime verification code, resulting in either fully synchronous or completely offline monitoring. By contrast the OPE is organised in terms of micro-services, making it relatively easy to have asynchronous monitoring on the live system: on the one hand, introducing the monitoring service was as straightforward as adding any other service to the system; while on the other hand, using the native communication infrastructure, all services can report events to the monitoring service.

Experience 4.3. Taken from ongoing project #1.

4.3 Challenge 3: Communication and Formalisation of Properties

One of the initial hurdles of introducing runtime verification in industry is that of expressing the system properties in a formal fashion. To address this problem, we are working on two fronts:

Using a controlled natural language One way of easing the difficulty of expressing correctness properties is by providing a specification language which does not require its users to have a background in formal methods. While automata have been useful in previous experiences, their expressivity is substantially limited except through the use of additional Java code on the transitions. One way of lifting this limitation without impinging on the understandability of the language is through the use of controlled natural languages [33]. These allow the creation of a custom language whose expressivity matches that required in the context while the learning curve can be kept to a minimum. We have experimented with the use of controlled natural languages in such contexts in more academic projects [12, 21, 14] before, but the OPE project (Experience 4.4) was the first industrial-project setting in which we have used this approach, and which has so far proved to be effective.

Generating monitors automatically Another approach being explored to simplify property specification is to attempt to extract them automatically or otherwise from available information already present in tests. There has been some previous work on automated monitor synthesis from tests e.g. [27, 2], although these approaches work at a level of abstraction which is not always available in real-life case studies. For instance, [27] requires model-based test case generators which are infrequently used in industry. There is some initial work to start from the (universally used in industry) unit tests, but it is still unclear how much can be achieved automatically. On the other hand, as a means of supporting manual property writing, there is no denying that tests contain much information which can be used for property writing (Experience 4.5).

Developing a controlled natural language
The OPE project is concerned with alleviating the administrative burdens of creating financial applications. As such, there are various laws and directives which need to be taken into consideration (e.g., the Electronic Money Directive and the Anti-Money Laundering Regulations). The main challenge with encoding such legislation into formal properties is that they are regularly updated, and that lawyers need to be involved to confirm that what is being specified corresponds to the law. Using a controlled natural language enabled us to have a communication language with the non-technical lawyers, and at the same time technical people would not need to be involved each time the legislation is updated.

Experience 4.4. Taken from ongoing project #1.

Generating monitors automatically
While none of our industrial partners had been using runtime verification before our collaboration, they both had a formidable test suite with good coverage of the system's functionality. This realisation led us to consider extracting monitors from tests. While the investigation is still in its early phases, initial experiments using the Daikon invariant inference engine suggest that a number of properties can indeed be extracted from tests automatically: depending on some quality attributes of the test suite such as branch coverage, we were able to exceed 70% specification recall, although admittedly precision is still below 30% [17].

Experience 4.5. Taken from ongoing project #2.

5 Conclusions

In this chapter we have presented an anecdotal view of the use of runtime verification in an industrial setting. Although we focussed on our experiences in the domain of financial transaction systems, much of the observations are not domain-specific, and can be extrapolated for other application domains.

The challenges encountered can mostly be split into two categories — firstly how runtime verification can be fitted into existing software engineering practices and management structures, and secondly technical ones, particularly tailoring the right runtime verification flavour to match the requirements and system at hand. We have found that some such choices tend to pave the way for smoother adoption of monitoring technologies — for instance, starting with offline verification using existing system behaviour logs can be an excellent way of showing potential benefit without having to surpass the hurdle of introducing new code into the system. Finally, we have identified the major challenges which we believe are still to be addressed before runtime verification can find a foothold in industry, enabling its widespread use.

References

1. Abela, P., Colombo, C., Pace, G.: Offline runtime verification with real-time properties: A case study. In: University of Malta Workshop in ICT (WICT'09) (2009)
2. Artho, C., Barringer, H., Goldberg, A., Havelund, K., Khurshid, S., Lowry, M.R., Pasareanu, C.S., Rosu, G., Sen, K., Visser, W., Washington, R.: Combining test case generation and runtime verification. Theor. Comput. Sci. 336(2-3), 209–234 (2005), http://dx.doi.org/10.1016/j.tcs.2004.11.007
3. Artho, C., Biere, A.: Combined static and dynamic analysis. Electr. Notes Theor. Comput. Sci. 131, 3–14 (2005), http://dx.doi.org/10.1016/j.entcs.2005.01.018
4. Azzopardi, S., Colombo, C., Pace, G.J.: A model-based approach to combining static and dynamic verification techniques. In: Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques - 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part I. pp. 416–430 (2016), http://dx.doi.org/10.1007/978-3-319-47166-2_29
5. Azzopardi, S., Colombo, C., Pace, G.J., Vella, B.: Compliance checking in the open payments ecosystem. In: Software Engineering and Formal Methods - 14th International Conference, SEFM 2016, Held as Part of STAF 2016, Vienna, Austria, July 4-8, 2016, Proceedings. pp. 337–343 (2016)
6. Baresi, L., Ghezzi, C.: The Disappearing Boundary Between Development-time and Run-time. In: Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research. pp. 17–22. FoSER'10, ACM, New York, NY, USA (2010), http://doi.acm.org/10.1145/1882362.1882367
7. Bartocci, E., Grosu, R., Karmarkar, A., Smolka, S.A., Stoller, S.D., Zadok, E., Seyster, J.: Adaptive runtime verification. In: Runtime Verification, Third International Conference, RV 2012, Istanbul, Turkey, September 25-28, 2012, Revised Selected Papers. pp. 168–182 (2012), http://dx.doi.org/10.1007/978-3-642-35632-2_18
8. Berkovich, S., Bonakdarpour, B., Fischmeister, S.: Gpu-based runtime verification. In: 27th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2013, Cambridge, MA, USA, May 20-24, 2013. pp. 1025–1036 (2013), http://dx.doi.org/10.1109/IPDPS.2013.105

9. Bodden, E., Hendren, L., Lam, P., Lhoták, O., Naeem, N.A.: Collaborative runtime verification with tracematches. J. Log. and Comput. 20(3), 707–723 (Jun 2010), http://dx.doi.org/10.1093/logcom/exn077
10. Bodden, E., Hendren, L., Lhoták, O.: A staged static program analysis to improve the performance of runtime monitoring. In: Proceedings of the 21st European Conference on Object-Oriented Programming. pp. 525–549. ECOOP'07, Springer-Verlag, Berlin, Heidelberg (2007), http://dl.acm.org/citation.cfm?id=2394758.2394793
11. Bodden, E., Lam, P., Hendren, L.J.: Clara: A framework for partially evaluating finite-state runtime monitors ahead of time. In: RV'10. LNCS, vol. 6418, pp. 183–197 (2010)
12. Calafato, A., Colombo, C., Pace, G.J.: A controlled natural language for tax fraud detection. In: Controlled Natural Language - 5th International Workshop, CNL 2016, Aberdeen, UK, July 25-27, 2016, Proceedings. pp. 1–12 (2016), http://dx.doi.org/10.1007/978-3-319-41498-0_1
13. Cauchi, A., Colombo, C., Francalanza, A., Micallef, M., Pace, G.: Using gherkin to extract tests and monitors for safer medical device interaction design. In: Proceedings of the 8th ACM SIGCHI Symposium on Engineering Interactive Computing Systems. pp. 275–280. EICS '16, ACM, New York, NY, USA (2016), http://doi.acm.org/10.1145/2933242.2935868
14. Cauchi, A., Colombo, C., Francalanza, A., Micallef, M., Pace, G.J.: Using gherkin to extract tests and monitors for safer medical device interaction design. In: Proceedings of the 8th ACM SIGCHI Symposium on Engineering Interactive Computing Systems, EICS 2016, Brussels, Belgium, June 21-24, 2016. pp. 275–280 (2016), http://doi.acm.org/10.1145/2933242.2935868
15. Centonze, P., Flynn, R.J., Pistoia, M.: Combining static and dynamic analysis for automatic identification of precise access-control policies. In: 23rd Annual Computer Security Applications Conference (ACSAC 2007), December 10-14, 2007, Miami Beach, Florida, USA. pp. 292–303 (2007), http://dx.doi.org/10.1109/ACSAC.2007.14
16. Chimento, J.M., Ahrendt, W., Pace, G.J., Schneider, G.: Starvoors: A tool for combined static and runtime verification of java. In: Runtime Verification - 6th International Conference, RV 2015 Vienna, Austria, September 22-25, 2015. Proceedings. pp. 297–305 (2015), http://dx.doi.org/10.1007/978-3-319-23820-3_21
17. Chircop, L., Colombo, C., Micallef, M.: Exploring the link between automatic specification inference and test suite quality (2017), submitted for publication
18. Colombo, C., Dimech, G., Francalanza, A.: Investigating instrumentation techniques for ESB runtime verification. In: Software Engineering and Formal Methods - 13th International Conference, SEFM 2015, York, UK, September 7-11, 2015. Proceedings. pp. 99–107 (2015), http://dx.doi.org/10.1007/978-3-319-22969-0_7
19. Colombo, C., Francalanza, A., Gatt, R.: Elarva: A monitoring tool for erlang. In: Runtime Verification - Second International Conference, RV 2011, San Francisco, CA, USA, September 27-30, 2011, Revised Selected Papers. pp. 370–374 (2011), http://dx.doi.org/10.1007/978-3-642-29860-8_29
20. Colombo, C., Francalanza, A., Mizzi, R., Pace, G.J.: polylarva: Runtime verification with configurable resource-aware monitoring boundaries. In: Software Engineering and Formal Methods - 10th International Conference, SEFM 2012, Thessaloniki, Greece, October 1-5, 2012. Proceedings. pp. 218–232 (2012), http://dx.doi.org/10.1007/978-3-642-33826-7_15
21. Colombo, C., Grech, J., Pace, G.J.: A controlled natural language for business intelligence monitoring. In: Natural Language Processing and Information Systems - 20th International Conference on Applications of Natural Language to Information Systems, NLDB 2015 Passau, Germany, June 17-19, 2015 Proceedings. pp. 300–306 (2015), http://dx.doi.org/10.1007/978-3-319-19581-0_27

22. Colombo, C., Pace, G.J.: Fast-forward runtime monitoring - an industrial case study. In: Runtime Verification, Third International Conference, RV 2012, Istanbul, Turkey, September 25-28, 2012, Revised Selected Papers. pp. 214–228 (2012), http://dx.doi.org/10.1007/978-3-642-35632-2_22
23. Colombo, C., Pace, G.J., Abela, P.: Compensation-aware runtime monitoring. In: Runtime Verification - First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings. pp. 214–228 (2010), http://dx.doi.org/10.1007/978-3-642-16612-9_17
24. Colombo, C., Pace, G.J., Abela, P.: Safer asynchronous runtime monitoring using compensations. Formal Methods in System Design 41(3), 269–294 (2012), http://dx.doi.org/10.1007/s10703-012-0142-8
25. Colombo, C., Pace, G.J., Schneider, G.: Dynamic event-based runtime monitoring of real-time and contextual properties. In: Formal Methods for Industrial Critical Systems, 13th International Workshop, FMICS 2008, L'Aquila, Italy, September 15-16, 2008, Revised Selected Papers. pp. 135–149 (2008), http://dx.doi.org/10.1007/978-3-642-03240-0_13
26. Colombo, C., Pace, G.J., Schneider, G.: LARVA — safer monitoring of real-time java programs (tool paper). In: Seventh IEEE International Conference on Software Engineering and Formal Methods, SEFM 2009, Hanoi, Vietnam, 23-27 November 2009. pp. 33–37 (2009), http://dx.doi.org/10.1109/SEFM.2009.13
27. Falzon, K., Pace, G.J.: Combining testing and runtime verification techniques. In: Model-Based Methodologies for Pervasive and Embedded Software, 8th International Workshop, MOMPES 2012, Essen, Germany, September 4, 2012. Revised Papers. pp. 38–57 (2012), http://dx.doi.org/10.1007/978-3-642-38209-3_3
28. Giannakopoulou, D., Pasareanu, C.S., Lowry, M., Washington, R.: Lifecycle Verification of the NASA Ames K9 Rover Executive. Tech. rep., NASA (2004), available from `http://ti.arc.nasa.gov/publications`
29. Hoare, C.: Programming is an engineering profession. In: Wallis, P. (ed.) State of the Art Report 11, No. 3: Software Engineering. pp. 77–84. Pergamon/Infotech (1983), also Oxford PRG Monograph No. 27.; and IEEE Software 1(2)
30. Hu, R., Yoshida, N.: Hybrid session verification through endpoint api generation. In: 19th International Conference on Fundamental Approaches to Software Engineering. LNCS, Springer (2016)
31. Jaksic, S., Bartocci, E., Grosu, R., Kloibhofer, R., Nguyen, T., Nickovic, D.: From signal temporal logic to FPGA monitors. In: 13. ACM/IEEE International Conference on Formal Methods and Models for Codesign, MEMOCODE 2015, Austin, TX, USA, September 21-23, 2015. pp. 218–227 (2015), http://dx.doi.org/10.1109/MEMCOD.2015.7340489
32. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J., Irwin, J.: Aspect-oriented programming. In: ECOOP. pp. 220–242 (1997), http://dx.doi.org/10.1007/BFb0053381
33. Kuhn, T.: A survey and classification of controlled natural languages. CoRR abs/1507.01701 (2015), http://arxiv.org/abs/1507.01701
34. Medhat, R., Bonakdarpour, B., Fischmeister, S., Joshi, Y.: Accelerated runtime verification of LTL specifications with counting semantics. In: Runtime Verification - 16th International Conference, RV 2016, Madrid, Spain, September 23-30, 2016, Proceedings. pp. 251–267 (2016), http://dx.doi.org/10.1007/978-3-319-46982-9_16
35. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977. pp. 46–57 (1977), http://dx.doi.org/10.1109/SFCS.1977.32
36. Smaragdakis, Y., Csallner, C.: Combining static and dynamic reasoning for bug detection. In: Tests and Proofs, First International Conference, TAP 2007, Zurich, Switzerland, February 12-13, 2007. Revised Papers. pp. 1–16 (2007), http://dx.doi.org/10.1007/978-3-540-73770-4_1

37. Tamura, G., Villegas, N.M., Müller, H., Sousa, J.a., Becker, B., Karsai, G., Mankovskii, S., Pezzè, M., Schäfer, W., Tahvildari, L., Wong, K.: Towards Practical Runtime Verification and Validation of Self-Adaptive Software Systems. In: de Lemos, R., Giese, H., Müller, H., Shaw, M. (eds.) Software Engineering for Self-Adaptive Systems II, Lecture Notes in Computer Science (LNCS), vol. 7475, pp. 108–132. Springer (January 2013), http://dx.doi.org/10.1007/978-3-642-35813-5-5
38. Wonisch, D., Schremmer, A., Wehrheim, H.: Zero Overhead Runtime Monitoring. In: SEFM'13, LNCS, vol. 8137, pp. 244–258. Springer Berlin Heidelberg (2013), http://dx.doi.org/10.1007/978-3-642-40561-7_17