Runtime Verification using VALOUR[*]

Shaun Azzopardi[1], Christian Colombo[1], Jean-Paul Ebejer[2], Edward Mallia[1],
and Gordon J. Pace[1]

[1] Department of Computer Science, University of Malta, Malta.
[2] Centre for Molecular Medicine and Biobanking, University of Malta, Malta.

Abstract

In this paper we give an overview of VALOUR, a runtime verification tool which has been developed in the context of a project to act as a backend verification tool for financial transaction software. A VALOUR script is written by the user and is then compiled into a verification system. Although, developed as part of a project, the tool has been designed as a stand-alone general-purpose verification engine with a particular emphasis on event consumption. The strong points of VALOUR when compared to other runtime verification tools is its focus on scalability and robustness.

1 Introduction

When adopting runtime verification technologies to improve the dependability of real-life systems, one major challenge is that the technology is deployed post-deployment, and thus, dependability of the runtime verification tool is essential. Given that the verification tools will be used at runtime and interacting with the actual system users would expect that the dependability of the combined verification tool and system-under-scrutiny should not be any lower than that of the original system. In order to convince users to adopt such tools, the confidence in the reliability of the runtime verification tool has to far exceed that of the system-under-scrutiny[1]. Testing, offline verification and static analysis tools have the luxury of being used prior to deployment, thus mitigating part of this problem. However, with online runtime verification, in which verification is performed during execution, this becomes an inevitable issue.

Over the past two years, we have been working on techniques to integrate runtime verification technology into the Open Payments Ecosystem (OPE) [2], an industrial system developed by Ixaris Ltd., which is planned to handle high volumes of financial transactions across different user applications and financial institutions. Compliance to legislation and correctness are critical in this domain, and the risk of failure due to the runtime verification module had to be mitigated.

---

[1]It has been remarked that the major challenge in selling any form of insurance is that of convincing the client that things can go wrong. When the insurance policy itself may have loopholes, thus possibly rendering it useless, convincing clients to pay for that insurance becomes doubly hard.

The runtime verification tools we typically used in our previous projects (Larva [8] and pollyRV [6]) were developed in an academic setting and in an iterative manner — with waves of students and researchers adding and changing features over the years[2]. Before the OPE project, we already had experience with Ixaris Ltd. on integrating monitoring into their systems e.g. [7], where the monitoring code was usually hand-coded for particular instances and manually instrumented into the system in order to avoid problems from arising. In the context of the OPE, it was decided, however, that a general-purpose runtime verification tool should be rebuilt from scratch — with just about sufficient features for the required functionality, and developed in a more software-engineering robust manner. An extenuating factor was also that the monitoring component of the OPE will be used in a microservice architecture, with most of the functionality residing as a separate independent service, thus reducing the severity of the dependence on it.

In this paper we give a brief overview of the runtime verification tool Valour which arose in this project. The architecture and design of the tool are presented in Section 2, while the specification language and features of the concrete syntax are presented in Section 3. The role of Valour in the OPE is described in Section 4.

2 Valour: Architecture and Design

Valour has been designed to work as a tool which processes streams of data received from (potentially) different sources. This is achieved in three stages: (i) a *trigger manager* processes triggers arriving from the systems being monitored, categorising them and tagging them appropriately with relevant data to package them as *events*; (ii) an *event manager* which consumes the events produced by the *trigger manager* farming them to the appropriate monitoring unit and triggering new ones if required; (iii) the *monitoring unit instances* which include the logic as to how to react upon receiving a relevant event. Since Valour allows for properties to be monitored for each instance of a particular type, it includes a *template instantiator* which creates a new monitoring unit whenever events pertaining to a new instance are received. The architecture of Valour is shown in Figure 1.

2.1 Events

At its core, Valour is an event processing system, with events corresponding to temporal points-of-interest occuring during the execution of the underlying systems. These events are annotated with additional information (e.g. parameters) and abstract away unnecessary concrete detail (e.g. the structure of the method call from which the event was extracted).

In Valour, events are named and can be parametrised by a data tuple e.g. *closeAccount(destinationAccount)* corresponds to an event named *closeAccount* carrying data parameter *destinationAccount* — corresponding to the account to which any remaining funds are being requested to be transferred. At the event level of abstraction we do not care whether this event comes from a method call in the main system (e.g. instrumented using aspect-oriented programming), from an intercepted update of the underlying database (e.g. instrumented using database triggers) or any other source — these correspond to the *triggers* from which the events are synthesised.

---

[2]As with many academic tools, the development of Larva and pollyRV lacked software engineering rigour — their code is poorly documented and in some places poorly structured and they lack a proper test suite. Furthermore, having acted as vehicles for feature experimentation, they are bloated with code and features which are not required for the core functionality and their use on real-life systems. The result is that it is difficult to convince industrial partners to use these tools on live systems.
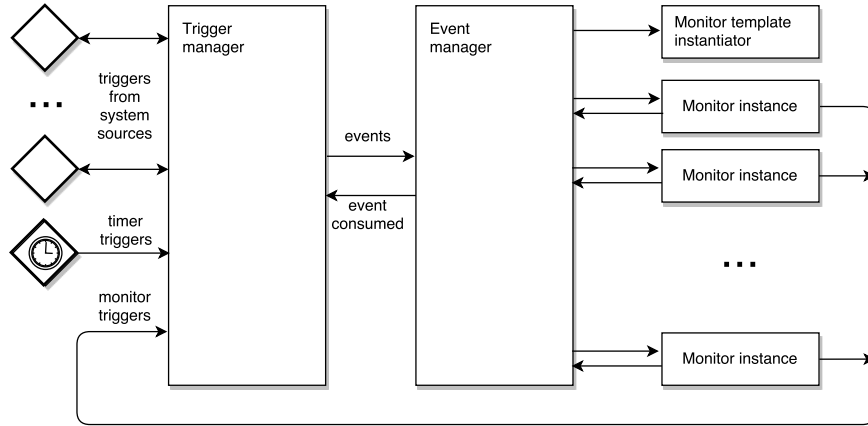
Figure 1: Architecture of Valour

In addition to parameters, events may be annotated by the entities or categories they correspond to. For instance, *closeAccount* may be categorised by the class of bank accounts (and thus relating every instance of the event to a specific account), or the category of users (and thus relating such events with the user who owns the account). This information, together with information of how to extract the category instance from the event, is used by Valour so that the event is passed on to the relevant monitor. For instance, if there are monitors checking compliance to the expected user life cycle, a *closeAccount* categorised to belong to the class of users is automatically passed onto the monitor of the related user.

2.2   Triggers: Event Generators

Valour makes a distinction between events, which are what the monitors can consume, and the raw *triggers* from which events are produced. Valour can handle (i) triggers coming from the underlying system(s); (ii) triggers generated by the monitoring logic; and (iii) triggers dependant on timers, firing on timeouts[3].

System triggers can come from different sources, and with different instrumentation approaches. Currently, Valour provides two constructs for defining such triggers:

- *controlflow trigger*: triggers corresponding to control points in the code of (a part of) the system, and would be instrumented via aspect-oriented techniques.

- *external trigger*: allow Valour users to define "custom" external triggers through an API which in turn can be consumed by Valour and translated into events. *External trigger* allows for integrating with other systems such as (i) message brokers to receive publish-subscribe messages; (ii) databases to identify changes to key entities; (iii) filesystems to watch files or directories; etc.

The separation of triggers from events helps keep the monitoring and verification specifications independent of the more technology-specific aspect of identifying points of interest during a system's execution. Thus Valour is easily extensible both internally, i.e. by creating new language constructs for new trigger types (e.g. new trigger construct to expose an HTTP interface

---

[3]At the time of writing, timer triggers are only partially supported by Valour.

to receive webhooks), and externally i.e. by allowing Valour users to create new integrations by implementing new *external trigger*s.

This was corroborated by our experience of adapting the runtime verification tool pollyRV (formerly known as PolyLarva) [6], where it was realised that the technology used by or in which the monitored systems were written, but also the type of specifications which were to be monitored, greatly changed the nature of the points-of-interest one had to capture. Whereas in imperative and object-oriented languages such as Java and C, control-flow points (e.g. entry into a method call) was usually desired, in a distributed system written in a language like Erlang, a new process being launched or dying were sometimes more important [5], and in a case study using PHP, file inclusion points were the crucial points-of-interest. In Valour, we have a flexible way in which triggers may be associated with different types of runtime points-of-interest by allowing for different tools to be used to instrument the trigger listeners.

2.3 Monitors: Event Consumers

Monitors in Valour are written in a specification format using a guarded command language with rules of the form $e \mid c \mapsto a$, which trigger when event e is produced, with boolean condition c holding, upon which action a (a chunk of code) is executed. We refer to a group of such rules as a *monitor*. There are two key features which enable the use of such rules for complex specifications:

Monitor templates: Many specifications are not global ones, but are to be instantiated for all instances of a particular category which might arise at runtime. In Valour this is supported by having monitor templates which quantify over a particular category of events, and are instantiated whenever a relevant event (i.e. one which is used by the monitor) belonging to a new instance of that category is received. Internally, Valour ensures that events are only passed to the relevant monitors, thus reducing overheads in event consumption.

Monitoring state: Monitors may carry local state to keep track of information relevant to a particular rule block. This state can be accessed through the rules' conditions and actions. For instance, a rule might be used to keep track of the running total of emoney transactions within an application, or to keep track of all users who transferred money to a particular geographical region over the past 24 hours.

These two features are particularly useful when combined together, with the state being replicated for each instance of a template. Thus, for example, each instance of a bank account object may be associated with a state to keep track of the amount of money transferred in and out of the account. Furthermore, nesting of templates and monitoring state can be used to structure specifications and ensuring locality of state. For instance, a monitor may be instantiated for each active user on a financial institution website, keeping track of running totals for each user, but then, within each user, launch a separate monitor for each transaction, ensuring it is done in a timely manner and without excessive charges.

2.4 Event Consumption

Valour supports both *synchronous* or *asynchronous* monitoring. Rather than giving the user a choice between globally performing all monitoring in one mode or another, we provide a more fine-grained approach, and the choice between the monitoring modes is at an event level. Each event is marked as requiring synchronous or asynchronous consumption, effectively instructing

the *Event Manager* how to present monitoring results when the triggers generating the event are observed at runtime — with synchronous events, the manager holds off supplying the result of the event consumption until all computation is completed, while in the case of asynchronous events, an intermediate acknowledgement is immediately sent back, which can be used to check for completion status and monitor results. The choice to provide such a fine grained approach arose when we observed that (i) in practice, not all properties are urgent enough to warrant paying the high cost of fully synchronous monitoring[4], and thus having some properties being monitored synchronously and others asynchronously can be a good idea; and (ii) certain events appear in critical parts of the system or have high occurrence density and blocking the system while consuming them would be detrimental. It is worth noting that the same trigger (e.g. a method call) can generate multiple events, thus allowing for it to be consumed synchronously in one property, and asynchronously in another.

Given that the specification language we adopted is effectively a set of rules which could, conceptually be consumed concurrently, it is worth noting that in some cases, sequentialisation is required since shared state between rules could otherwise lead to race conditions. When the replicated monitors have a disjoint state and could thus be safely monitored concurrently, we support a separate replication construct to allow for this[5]. Using this quantifier, the Valour script writer takes on the responsibility of ensuring that concurrent state access while Valour invokes nested monitors is safe. Unless so specified, Valour consumes events sequentially to ensure determinism.

3   Valour Syntax

Valour scripts are split into two main parts. In the first, declaration part of the script, categories are declared and events are defined in terms of triggers. In the second part, monitors are then defined over these events.

Events are defined by relating them to a relevant trigger. For example, in the following example, the event userLogin() is associated with the system control-flow trigger firing when the successfulLogin method is called:

```
event userLogin() = {
  system controlflow trigger LoginHandler.successfulLogin(User u)
}
```

Events can be enriched with parameters which are related to the system method parameters directly or computed in a where clause of the event definition:

```
event transferSubmitted(Long userId, BigDecimal amount) = {
  system controlflow trigger
    TransactionHandler.submitTransfer
      (User u, String toAccount, BigDecimal amount)
  where
    userId = { u.getId() }
}
```

In the declaration part of the script, events can be declared to belong to these categories: an event is created with the value of the *getId* being supplied by the trigger through object *u*.

---

[4]We are assuming that the monitoring is performed on a separate address space, and thus asynchronous monitoring will have a much lower impact on the system's performance.

[5]At the time of writing, *par-for-each* is still being integrated.

```
category USER indexed by Long

event userLogin(u) = {
  system controlflow trigger LoginHandler.successfulLogin(User u)
  belonging to USER
    with index { u.getId() }
}
```

This ensures that if the `userLogin` event is used in a monitor quantified over the category USER, any runtime appearance of the event is associated with the monitor indexed as specified in the event declaration. If the index has not been encountered before, a new instance of the monitor is created with the index.

Monitors are built from rules of the form: `event | condition -> action`, where the condition and action can be arbitrary Xtend[6] code to be executed. Monitoring state can be added by declaring variables in the first block of the `state {...} in {...}` structure — variables which can be used in the conditions and actions of the second block. Below is a monitoring block which keeps count of users logged in, and generates an alarm whenever a normal user logs in while there are at least 100 users already logged in:

```
state {
  Integer userCount = { 0 }
} in {
  userLogin(u) | { u.isGoldUser() } -> { userCount++; }
  userLogin(u) | { u.isNormalUser() && userCount < 100 } -> { userCount++; }
  userLogin(u) | { u.isNormalUser() && userCount >= 100 }
    -> { generateAlarm("User saturation level exceeded"); }
  ...
}
```

Monitors are quantified over a category defining monitor templates using the `replicate { state } foreach category {rules}`, in which the rules are replicated for each instance of the category, each containing a copy of the state. Below is an example of such a replicated monitor:

```
replicate {
  BigDecimal ownTransactionsTotal = { 0 }
} foreach USER u {
  transfer(src, dst, value)
    |  { dst.owner() == u }
    -> { ownTransactionsTotal += value; }
}
```

A more complete description of the syntax can be found in the Valour user manual [4].

4   Real-Life use of Valour

Ixaris Systems Ltd[7] recently launched the Open Payments Ecosystem (OPE), in a move to make financial transactions more accessible to the developer community. The framework connects

---

[6]Xtend is a general-purpose high-level programming language for the Java Virtual Machine (http://www.eclipse.org/xtend/). Xtend is considered as an expressive dialect of Java which translates into readable Java source code.

[7]http://www.ixaris.com

service providers (SPs) and developers with programme managers (PMs) wishing to deploy applications offering payment services. Given the high level of risk, and regulatory oversight involved when offering such services, SPs are often unwilling to adopt applications by developers they do not trust. Therefore, regulatory compliance checking is an integral part of the OPE, intended to give SPs information relevant to their decision to adopt (or not) an application, and confidence in that the application is compliant at runtime (or that violations will be promptly detected).

Through an iterative process with payment services law experts and OPE developers, we identified twenty-three regulations relevant to the OPE. Twelve of these can be verified statically at compile-time (see [1, 2] for more information), while eleven are done at runtime, i.e. checked using Valour. An example of such a rule is *For each card i and payment application p, where i is a card of p, p is regulated in the UK, and i has expired less than 12 months ago, then e-money in i is redeemed without fees*, which was transformed into a script as follows:

```
startOfRedemption(programmeCountry, expiryDate, fee)
  | { fee != 0 &&
      "UK".equals(programmeCountry) &&
      Period.between(expiryDate, LocalDate.now()).toTotalMonths < 12
    }
  -> { error("Emoney Regulation 39 violated."); }
```

Other examples maintain some state (e.g. one rule sets a limit on the cumulative amount that can be redeemed from a card if no full due diligence is performed). The rules quantify over at most one object, with three rules maintaining state between transaction start and end, one maintaining state at the card level (i.e. monitoring all transactions involving that card, given some condition) and another for a whole programme (i.e. all transactions involving cards in the programme potentially trigger the rule). The state kept is not very memory-intensive, and only involves counters and timestamps. By default Valour uses an in-memory store, but custom stores (e.g. a MySQL database store) can be plugged in as needed. On a violation, the above example reports the regulation violated as a string for brevity. In the actual implementation, a JSON string is passed with more information (e.g. the relevant transaction and programme identifiers). In general Valour provides an extensible return value, allowing the user to design their own return type.

Valour acts as a microservice within the OPE[8], with the OPE both triggering directly the engine and waiting for a response, and publishing a transaction event trigger passively. When a transaction is attempted, the OPE always requests the engine to check it for compliance. If a certain amount of time elapses (milliseconds) without a response, the OPE proceeds and Valour, instead of responding directly, publishes an event in case of a violation. Although Valour is not expected to skip this time bound for simple rules, however there are more complex rules for which more information needs to be requested (e.g. about a card's owner) which depend on the response time of the OPE.

We are currently approaching the testing phase of this usage of Valour, with integration between the OPE and the compliance engine being mostly completed. Another use of Valour identified within the OPE, but not yet implemented, is as a monitoring and reporting tool for risk. Service providers would be able to specify a risk appetite profile (e.g. they may not be willing to support transactions over a certain amount from a high risk country), which monitors can enforce and report on the risk of a programme at runtime.

---

[8]Since Valour is part of a sensitive system, we have so far decided not to release it.

5   Conclusions

In this paper, we have presented the runtime verification tool Valour. The tool has been integrated as part of a real-life financial transaction system in order to verify and monitor various elements, including legal compliance verification, risk-analysis for banks and service providers on the OPE, and resource-flow analysis of the applications using the OPE to perform their financial transactions. It is worth noting that although the the tool is already production ready, in that it is already being used on an industrial system, and is usable in practice by other parties, it is still under active development and evaluation, and certain features are still partially available.

We are still conducting evaluation of the reliability and robustness of Valour. However, the fact that it was developed using a rigorous software engineering regime and is thus better documented and tested than our previous tools, ensured it was more readily adopted for use within a critical real-life system. Furthermore, its use in the heavily tested environment of the OPE has ensured that many potential issues with the system have been identified and addressed.

In terms of features, Valour has borrowed many features from other runtime verification tools. For example, it supports parametrised traces using an approach similar to the way in which Larva [8], MarQ [10], and JavaMOP [9]; it supports a hybrid synchronous and asynchronous approach to event consumption as done by detectEr [3] and SMEDL [11]; it supports a degree of technology agnosticism, allowing it to be (relatively) easily extended to support other programming languages as does pollyRV [6]. However, the primary observation arising from our experience of developing Valour remains that there is still limited work in software engineering practice to support the development of runtime tools aimed at increasing dependability of an underlying system.

References

[1] Shaun Azzopardi, Christian Colombo, and Gordon J. Pace. 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*, pages 416–430, 2016.

[2] Shaun Azzopardi, Christian Colombo, Gordon J. Pace, and Brian Vella. 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*, pages 337–343, 2016.

[3] Ian Cassar and Adrian Francalanza. On synchronous and asynchronous monitor instrumentation for actor-based systems. In *Proceedings 13th International Workshop on Foundations of Coordination Languages and Self-Adaptive Systems, FOCLASA 2014, Rome, Italy, 6th September 2014.*, pages 54–68, 2014.

[4] Edward Mallia Christian Colombo, Jean Paul Ebejer and Gordon J. Pace. Valour homepage. University of Malta, 2017. http://www.cs.um.edu.mt/svrg/Tools/Valour/.

[5] Christian Colombo, Adrian Francalanza, and Rudolph Gatt. Elarva: A monitoring tool for erlang. In *Runtime Verification - Second International Conference, (RV)*, volume 7186 of *Lecture Notes in Computer Science*, pages 370–374. Springer, 2012.

[6] Christian Colombo, Adrian Francalanza, Ruth Mizzi, and Gordon J. Pace. 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*, pages 218–232, 2012.

[7] Christian Colombo and Gordon J. Pace. 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*, pages 214–228, 2012.

[8] Christian Colombo, Gordon J. Pace, and Gerardo Schneider. 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*, pages 33–37, 2009.

[9] Patrick O'Neil Meredith, Dongyun Jin, Dennis Griffith, Feng Chen, and Grigore Rosu. An overview of the MOP runtime verification framework. *STTT*, 14(3):249–289, 2012.

[10] Giles Reger. An overview of marq. In *Runtime Verification - 16th International Conference, RV 2016, Madrid, Spain, September 23-30, 2016, Proceedings*, pages 498–503, 2016.

[11] Teng Zhang, Peter Gebhard, and Oleg Sokolsky. SMEDL: combining synchronous and asynchronous monitoring. In *Runtime Verification - 16th International Conference, RV 2016, Madrid, Spain, September 23-30, 2016, Proceedings*, pages 482–490, 2016.