Runtime Verification using Larva

Christian Colombo and Gordon J. Pace

Department of Computer Science, University of Malta, Malta.

Abstract. Larva, which has been in use and continuous development for almost a decade, has been extended in several ways and used in a wide range of scenarios, from industrial deployment to educational ones. In this paper we give an overview of Larva and give an overview of its extensions and uses.

1 Introduction

Larva [13, 14] was originally developed by the authors between 2007 and 2008 with the main drive being that of an industrial financial transaction system guiding the choice of features and design of the tool. Since then, the tool has been used (i) in various industrial projects and case studies; (ii) as a test-bed (to experiment with extensions and re-implementations) for research purposes; (iii) for teaching purposes. In the rest of the paper, after an overview of Larva in the next section, we dedicate a section to each of these three aspects. Finally, we relate Larva to other tools and conclude.

2 High-Level Overview of the Tool

Perhaps the most defining aspect of Larva is the use of symbolic automata as the basis of the specification language. This allows users who are already familiar to finite state machines to quickly grasp how to specify properties, while at the same time ensuring Turing completeness by allowing Java code to be embedded in the transitions. As an example consider the property below which keeps track of a risk value depending on a sequence of transaction actions carried out by the user. In particular note how by detecting relevant events, the automaton transitions from one state to another and updates a float value, *risk*, in the process. The label of the transition (given in square brackets) contains three backslash separated elements — the event which triggers the transition, a condition which must hold for the transition to be taken and the action to be executed when the transition is taken.

```
Variables {float risk;}
Property RiskManagement {
  States {...}
  Transitions {
    start  -> active [activate   \ \ risk=1;]
```

```
    active -> active [spendMoney \ \ risk*=0.5;]
    active -> active [createCard \ risk < 5 \ risk*=1.7;]
    active -> danger [createCard \ risk >= 5 \ ]
    ...
  }
}
```

Another defining element of LARVA is the *foreach* construct which allows top level universal quantification in specifications in a straightforward manner. Building on the previous example, using the `foreach` construct, the property can be instantiated for every unique user encountered[1]:

```
foreach (User u) {
  Variables {...}
  Property RiskManagement {...}
}
```

LARVA also supports the property engineer by providing timer actions on transitions. These make it easier to define real-time properties such as marking a user inactive following 30 days without carrying out any transaction (and then detecting a violation if any transaction is detected while inactive):

```
foreach (User u) {
  Variables {...
    Clock inactivity;
  }
  Property RiskManagement {
   States {...}
   Transitions {
     ...
     active -> active
       [u.spendMoney \ \ risk*=0.5; u.inactivity.reset();]
     active -> inactive [u.inactivity @ 30 days \ \]
     inactive -> violation [anyActivity \ \]
     ...
    }
  }
}
```

Finally, to facilitate the definition of properties which can benefit from a modular definition, LARVA properties can communicate between themselves through the use of non-blocking channels — allowing the transmission of any Java object across monitors over these internal communication events. In what follows, we extend the example such that the individual user's property sends updates to a central property which keeps count of high risk users:

[1] This effectively checks the property for every instance of class `User` using the default notion of object equivalence, although which notion of equivalence to use can be set by the property writer.

```
Variables {...
  int highRiskUsers=0;
}
Property systemRisk {...
  normalRisk -> normalRisk
      [notify.receive(u,risk) \ highRiskUsers < 100 \ highRiskUsers++;]
  normalRisk -> highRisk
       [notify.receive(u,risk) \ \ alertHuman();]
}

foreach (User u) {
  Property RiskManagement {
    ...
    active -> active [createCard \ risk > 5 \ notify.send(u,risk);]
    ...
  }
}
```

To support users in making the best use of the above extensive feature list, Larva is supported by a comprehensive user manual providing a running example, ships with a number of examples, and has also been more recently integrated into an Eclipse plugin[2] which provides syntax highlighting and automatic seamless generation of the monitoring (.lrv) files in an Eclipse project.

3 Case Studies

Larva has been conceived in the context of an industrial case study [13] in the financial transactions industry. Later, it was again applied and extended in several ways (see Section 4) on another, more extensive, case study in the same domain [10]. Subsequently, Larva has been applied to numerous other case studies from other domains including astronomy, user profiling, business intelligence, and video surveillance. To avoid repeating what has already been published, below we only give a short description:

Business intelligence through Facebook Keeping track of all the relevant comments, messages, and posts might be a daunting task for a marketing officer responsible for social media. Through the use of runtime monitoring this can be alleviated by enabling the user to specify rules in a purposely-designed language [9, 12].

Profiling user web interfaces Understanding how users use a web interface can be crucial in improving the design to help users complete their tasks efficiently. Through the use of runtime verification, voluminous logs of user actions on a web interface could be recognised efficiently as fitting into particular usage patterns and generating usage statistics [12].

[2] http://www.cs.um.edu.mt/svrg/Tools/LARVA/update-site/

Tax fraud detection Tax fraud experts would typically rely on technical personnel to carry out their querying of the data available. To avoid the communication overhead, with its hurdles of misunderstandings, a controlled natural language which automatically compiles into monitors has been designed and implemented [4].

Intelligent video surveillance Watching and analysing hours of surveillance footage is tedious for humans to do. One way of automating this is by allowing a user to specify rules which would classify suspicious behaviour. A case study has been carried out at a high security venue in Malta using this technique [12].

Astronomy Radio telescope observations result in large chunks of data which needs to be sifted for patterns of interest. While there are techniques in place to achieve this, runtime monitoring techniques provide another alternative [12].

Monitoring an enterprise service bus In component-based systems such as an enterprise service bus, where components can easily be added and removed, runtime monitoring provides more benefit over and above testing (than in the case of monolithic systems), since it is harder for testing to be representative all the possible environments a component would be functioning in. As a means of exploring ways in which runtime verification could be applied in this context, Larva has been instrumented in several ways together with the Mule Enterprise Service Bus [6].

Network intrusion detection Intrusion detection systems often rely on a number of rules which their users set to identify suspicious behaviour. A case study was carried out using duration calculus as a specification language, which was then translated automatically into native Larva automata [16].

4 Variations of Larva

Over the years, a number of extensions have been added to Larva with the aim of providing more features and make it more easily usable. We split this into different types of extensions: (i) extensions from an expressivity point of view, i.e., providing additional notations in which to express properties (but not extended expressivity *per se*), and (ii) architectural modifications which enable Larva to provide additional features.

Expressivity enhancement

Interval time logics While timer actions offer the property engineer a quick way of constructing monitors for real-time properties dealing with points in time, reasoning about time intervals might be more challenging. For example consider the property "*There should never be more than three bad logins in any one minute interval*". While it can be encoded using timers, it would be significantly straight forward had the notion of an interval been natively supported. In this respect, a Larva add-on[3] provides conversion of a subset

[3] http://www.cs.um.edu.mt/svrg/Tools/LARVA/Converter.zip

of duration calculus formulae [16] as well as QDDC formulae [15] into Larva notation.

Statistics Taking the view of monitor-oriented programming instead of strict runtime verification, we explored the possibility of using monitors to collate statistical data. In this respect, we created a notation extension [8] which supports two main additional constructs: one supporting point statistics and one supporting interval statistics. Point statistics are those which aggregate statistical results over the whole system history while interval statistics compute information over intervals defined in terms of the starting and ending events.

Domain specific languages Larva has also been used as an intermediate language to which domain specific languages can be compiled. Of particular significance are the following two: Firstly, we have compiled a language in the domain of business intelligence gathering from social media [9]. Secondly, we have compiled a language from the domain of tax fraud detection into Larva specifications [4].

Feature enhancement

Asynchronous monitoring Where one would want to keep the intrusion of monitoring on the system to a bare minimum, an option is to monitor asynchronously, i.e., allowing the monitor to lag behind the system. A version of Larva does this by consuming events from a database rather than aspect-oriented programming [10].

Database support When monitoring real-life industrial system, a considerable concern is to ensure the monitor behaves correctly even under a system crash, or when the resources required for monitoring grow significantly large. For this purpose, a version of Larva comes equipped with a database to store and retrieve monitors [10]. This means that in case of a system crash, the monitor state would not be lost, and the size of the monitor state would not be constrained by the main memory.

Monitor fast-forwarding Due to the monitors usually needing to keep some kind of state (essentially, a summary of the system's history), if monitors are substantially modified or new ones introduced, the monitor state might have to be rebuilt taking into consideration all of the system's history. One way of bypassing this is if the property writer can provide a way of directly abstracting away irrelevant details of the system's history to compute the monitor state at a particular point in time without replaying all the events individually, i.e., the point at which the new (or modified) monitors are introduced. This has been incorporated as a feature in one of the flavours of Larva [11].

Dynamic state generation In cases where the number of explicit monitor states is significantly high, the property engineer might prefer to encode such states programmatically, i.e., compute the next state program without providing an enumeration of all the possible states (similar to property monitoring through rewriting as opposed to full a priori state space exploration). The tool adaptation supporting this feature has been called dLarva [5].

Memory-bounded monitoring A major concern in monitoring is the resources used, since these are typically consumed from those available to the system. While this is virtually impossible to avoid, one way an engineer could control the effects this might have on the system is by being aware of a predetermined upperbound so that enough resources can be allocated by design. This is supported in LARVA [15] by enabling monitors to be defined as Lustre code for which computing strict resource consumption upperbounds is standard.

Combining LARVA with static analysis Overheads induced by runtime verification can be a concern in some systems. One way in which this has been addressed in the literature is by using static analysis to simplify the properties and reduce what still has to be checked at runtime. In STaRVOOrs[4] [1, 2], LARVA has been combined with the deductive verification tool KeY to verify properties which combine data-flow aspects (in the form of pre-/postconditions) and control-flow ones (in terms of LARVA automata).

We conclude this section by noting that while the main LARVA implementation is for Java (and all the above extensions are in Java), there are also basic implementations for Erlang [7], C#, and an adaptation for the (Java-based) OSGi framework [19].

5 LARVA in Education

Over the years, LARVA has been used for teaching runtime verification. Due to its automata-based notation, even undergraduate students with a knowledge of Java and automata can use the tool. Students are typically expected to be able to write and modify specifications in LARVA. The tool has also been used in graduate courses, in which students get to build their own simplified version of LARVA. They are given basic infrastructure code (for instance to parse properties), and through the course they add code to enable their system to mimic LARVA features.

6 Related Work and Conclusions

There are several tools which are close to LARVA in both their architecture and purpose, particularly, JavaMOP [17] (which came before LARVA) and MarQ [18] (which came after). What might be considered as the contribution of body of work surrounding LARVA is that it broke off from the traditional specification languages (particularly LTL) towards an automaton-based notation. Another interesting difference is that LARVA has always been used on case studies significantly different from what previous Java tools had been applied to: Before LARVA, Java properties revolved around correct API usage checks (e.g., correct action sequences on iterators [3]); on the other hand, the properties LARVA has been used for, are higher level "business rules" such as the one in the example above. The philosophy behind this choice was that if the Java API requires

[4] http://cse-212294.cse.chalmers.se/starvoors/

monitoring, then this should be something which ships with the JVM (which can perhaps be turned on and off), without involving the programmer in such concerns. Conversely, having a specification language which is high-level enough could even allow the quality assurance personnel to write the properties rather than the developers. This has the added benefit that the property writers are not the programmers — with more value for one view of the system validating the other.

While the ease with which one can write properties would be one of the pluses of Larva, it is not a tool which has been primarily designed for efficiency. However, typically when monitoring high level business rules, one would not expect events of interest to occur as frequently as when monitoring low level properties. Another element which has never been properly tackled in Larva is dealing with concurrency (and distribution). Although Larva ensures there are no data races by strictly avoiding non-determinism, the extremely prudent approach of serialising all threads leads to inefficiency.

Summary Summarising the above into the main characteristics of Larva:

+ Ideal where high property expressivity is preferred
+ Suitable for high-level properties
+ Low learning curve for non-logicians
– Not for applications where efficiency is crucial
– No support for dealing with concurrency or distribution efficiently

References

1. Wolfgang Ahrendt, Jesús Mauricio Chimento, Gordon J. Pace, and Gerardo Schneider. A specification language for static and runtime verification of data and control properties. In *FM 2015: Formal Methods - 20th International Symposium, Oslo, Norway, June 24-26, 2015, Proceedings*, pages 108–125, 2015.
2. Wolfgang Ahrendt, Jesús Mauricio Chimento, Gordon J. Pace, and Gerardo Schneider. Verifying data- and control-oriented properties combining static and runtime verification: theory and tools. *Formal Methods in System Design*, pages 1–66, 2017.
3. Eric Bodden, Feng Chen, and Grigore Rosu. Dependent advice: a general approach to optimizing history-based aspects. In *Aspect-oriented software development (AOSD)*, pages 3–14. ACM, 2009.
4. Aaron Calafato, Christian Colombo, and Gordon J. Pace. A controlled natural language for tax fraud detection. In *Controlled Natural Language - 5th International Workshop, CNL 2016, Aberdeen, UK, July 25-27, 2016, Proceedings*, pages 1–12, 2016.
5. John Paul Cassar, Christian Colombo, and Gordon J. Pace. Dynamic automata in larva. Technical Report 02-WICT-2010, Department of Computer Science, University of Malta, 2010.
6. Christian Colombo, Gabriel Dimech, and Adrian Francalanza. 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*, pages 99–107, 2015.

7. 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.
8. Christian Colombo, Andrew Gauci, and Gordon J. Pace. Larvastat: Monitoring of statistical properties. In *Runtime Verification - First International Conference, (RV)*, volume 6418 of *Lecture Notes in Computer Science*, pages 480–484. Springer, 2010.
9. Christian Colombo, Jean-Paul Grech, and Gordon J. Pace. 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*, pages 300–306, 2015.
10. Christian Colombo, Gordon Pace, and Patrick Abela. Safer asynchronous runtime monitoring using compensations. *Formal Methods in System Design*, 41(3):269–294, 2012.
11. Christian Colombo and Gordon J. Pace. Fast-forward runtime monitoring - an industrial case study. In *Runtime Verification, Third International Conference, RV 2012*, volume 7687 of *Lecture Notes in Computer Science*, pages 214–228. Springer, 2012.
12. Christian Colombo, Gordon J. Pace, Luke Camilleri, Claire Dimech, Reuben A. Farrugia, Jean-Paul Grech, Alessio Magro, Andrew C. Sammut, and Kristian Zarb Adami. Runtime verification for stream processing applications. In *Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications - 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece*, volume 9953 of *Lecture Notes in Computer Science*, pages 400–406, 2016.
13. Christian Colombo, Gordon J. Pace, and Gerardo Schneider. Dynamic event-based runtime monitoring of real-time and contextual properties. In *Formal Methods for Industrial Critical Systems (FMICS)*, volume 5596 of *Lecture Notes in Computer Science*, pages 135–149. Springer, 2008.
14. Christian Colombo, Gordon J. Pace, and Gerardo Schneider. Larva — safer monitoring of real-time java programs (tool paper). In *Software Engineering and Formal Methods (SEFM)*, pages 33–37. IEEE, 2009.
15. Christian Colombo, Gordon J. Pace, and Gerardo Schneider. Resource-bounded runtime verification of java programs with real-time properties. Technical Report CS2009-01, Department of Computer Science, University of Malta, 2009. Available from `http://www.cs.um.edu.mt/~reports`.
16. Christian Colombo, Gordon J. Pace, and Gerardo Schneider. Safe runtime verification of real-time properties. In *Formal Modeling and Analysis of Timed Systems, 7th International Conference (FORMATS)*, volume 5813 of *Lecture Notes in Computer Science*, pages 103–117, Budapest, Hungary, 2009.
17. Patrick O'Neil Meredith, Dongyun Jin, Dennis Griffith, Feng Chen, and Grigore Roşu. An overview of the MOP runtime verification framework. *International Journal on Software Techniques for Technology Transfer*, 14:249–289, 2012.
18. Giles Reger, Helena Cuenca Cruz, and David E. Rydeheard. Marq: Monitoring at runtime with QEA. In *Tools and Algorithms for the Construction and Analysis of Systems TACAS*, volume 9035 of *Lecture Notes in Computer Science*, pages 596–610. Springer, 2015.
19. Stéphane Frénot Christian Colombo Yufang Dan, Nicolas Stouls. A Monitoring Approach for Dynamic Service-Oriented Architecture Systems. In *SERVICE COMPUTATION 2012: The Fourth International Conferences on Advanced Service Computing*, pages 20–23. XPS (Xpert Publishing Services), 2012.