LARVA — Safer Monitoring of Real-Time Java Programs (Tool Paper)

Christian Colombo
Department of Computer Science
University of Malta, Malta
Email: christian.colombo@um.edu.mt

Gordon J. Pace
Department of Computer Science
University of Malta, Malta
Email: gordon.pace@um.edu.mt

Gerardo Schneider
IT University of Göteborg, Sweden
University of Oslo, Norway
Email: gersch@ituniv.se

***Abstract*—The use of runtime verification, as a lightweight approach to guarantee properties of systems, has been increasingly employed on real-life software. In this paper, we present the tool LARVA, for the runtime verification of properties of Java programs, including real-time properties. Properties can be expressed in a number of notations, including timed-automata enriched with stopwatches, Lustre, and a subset of the duration calculus. The tool has been successfully used on a number of case-studies, including an industrial system handling financial transactions. LARVA also performs analysis of real-time properties, to calculate, if possible, an upper-bound on the memory and temporal overheads induced by monitoring. Moreover, through property analysis, LARVA assesses the impact of slowing down the system through monitoring, on the satisfaction of the properties.**

I. INTRODUCTION

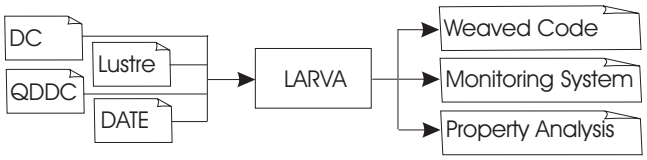
A growing area in formal methods is runtime verification — the monitoring of the program being executed by verifying the generated events against a set of properties. A particularly challenging aspect is the monitoring of real-time properties. Apart from being difficult to express and monitor, an additional challenge in introducing real-time, is that they are not always invariant under monitoring. Monitoring introduces overheads over and above the system, which may have the side effect of affecting the validity of the properties. In our approach, we have created a runtime verification architecture called LARVA.[1] This enables the specification of properties, including real-time, and the monitoring of Java programs against the specified properties. The tool also performs property analysis to give feedback regarding the effect of monitoring the given properties. The tool has been applied to a number of case studies, including a real-life financial system handling credit card transactions.

II. LARVA

A runtime verification architecture normally involves the following five components: (i) a system to be monitored; (ii) a set of specifications written in some formal notation; (iii) a stream of events extracted from the system in (i); (iv) a monitoring system which receives the events and verifies them according to the specification in (ii); and (v) possibly a feedback loop. The LARVA architecture is no exception and has the above five components.

A user who wants to monitor a system using LARVA must supply the system itself — a Java program — and a set of specifications in the form of a LARVA script — a textual representation of DATEs [1], similar to timed-automata enriched with stopwatches. Using the LARVA compiler the specification is transformed into the equivalent monitoring code, together with a number of aspects that extract the events from the system. Aspects are generated in AspectJ, one of the aspect-oriented implementations for Java, enabling automatic code injection without directly altering the actual code of the system. In LARVA, apart from extracting events, aspects are also used to send feedback to the system. Note that only Java byte code is necessary for instrumentation, thus LARVA can monitor third-party software. However, the author of the properties requires some knowledge of the system source code since most events in a LARVA script include method names.

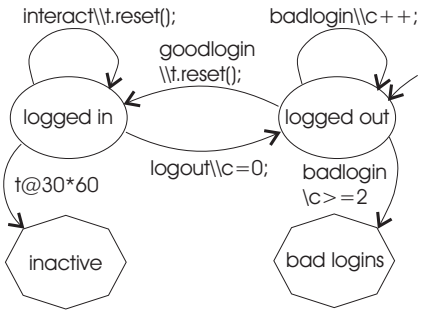
Although the 'native' logic of LARVA is DATE, the tool allows for properties to be written in a number of other logics (which are internally translated into DATEs) for runtime monitoring. These specification languages and logics include QDDC [2], Lustre [3] and a subset of the duration calculus called counterexample traces [4]. The complete architecture is shown below:



As an example, consider a system where one needs to monitor bad logins and the activity of a logged in user. By having access to *badlogin*, *goodlogin* and *interact* events (each of which corresponds to a method call in the Java program), one can keep a successive bad-login counter and a clock to measure the time a user is inactive. Fig. 1 shows the specification of a property stating that there are no more than two successive bad logins and 30 minutes of inactivity when logged in, expressed as a DATE automaton [1]. Transitions have three (backslash separated) labels: (i) the event triggering

The research work disclosed in this publication is partially funded by Malta Government Scholarship Scheme grant number ME 367/07/29 and by the Malta National Research and Innovation (R&I) Programme 2008 project number 052.

[1]The LARVA system, including further documentation and examples, is available from `http://www.cs.um.edu.mt/~svrg/Tools/LARVA`.




```
GLOBAL {
 VARIABLES {
   int c = 0;
   Clock t;
 }
 EVENTS {
   badlogin() = {*.badlogin()}
   clk() = {t@30*60}
   ...
 }
 PROPERTY users {
   STATES {
     BAD { badlogins inactive }
     NORMAL { loggedin }
     STARTING { loggedout }
   }
   TRANSITIONS {
     loggedout -> badlogins [badlogin\c>=2\]
     loggedout -> loggedin  [goodlogin\\t.reset();]
     ...
     loggedout -> loggedout [badlogin\\c++;]
     loggedin -> inactive   [clk\\]
   }
 }
}
```

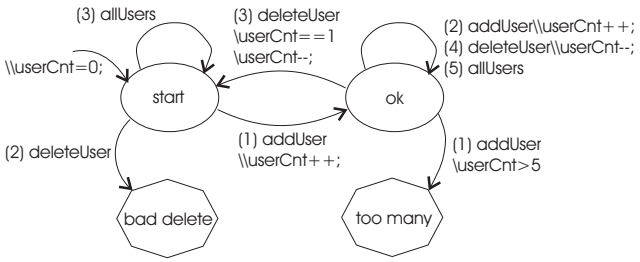
Fig. 1. The automaton and LARVA code of the bad logins scenario.

it; (ii) the condition which is checked before taking it; and (iii) the action performed when it is taken. A total ordering on the transitions is used to ensure determinism.

Furthermore, one may have properties which must hold for every user in a bank, or possibly properties which should hold for each account owned by each user. Consider the monitoring of a simplified banking system, in which we would want to monitor that at any time there should never be more than five users in the bank and that a deletion does not occur when there are no users (see Fig. 2).

To apply the above logic for each user and limit each user's number of accounts, one would simply need to replace *GLOBAL* with *FOREACH* in the first line of the code shown in Fig. 2, and apply the logic to accounts instead of users as follows:

```
FOREACH (User u) {
  ...
  PROPERTY accts {
    ...
    TRANSITIONS {
      start -> ok [addAcct()\\userAcct++;]
      ...
      ok -> ok [delAcct()\\userAcct--;]
    }
```




```
GLOBAL {
 VARIABLES {
   int userCnt = 0;
 }
 EVENTS {
   addUser() = {*.addUser()}
   delUser() = {*.deleteUser()}
   allUsers() = {User u.*()}
 }
 PROPERTY users {
   STATES {
     BAD { toomany baddel }
     NORMAL { ok }
     STARTING { start }
   }
   TRANSITIONS {
     start -> ok [addUser()\\userCnt++;]
     start -> baddel [delUser()\\]
     ...
     ok -> ok [delUser()\\userCnt--;]
     ok -> ok [allUsers()]
   }
 }
}
```

Fig. 2. The automaton and LARVA code of the simplified bank system.

A. Key Features of the Tool

LARVA provides three distinctive features: (1) its highly expressive logic, (2) its ability to calculate time and memory overheads, and (3) its guarantee on real-time properties. In what follows we explain each one of these features thoroughly.

1) A highly expressive logic: LARVA can be used to monitor all properties which can be written in DATEs. In other work we have given an in-depth account of the expressivity of DATEs and compared it to the expressivity of other similar tools [1]. In this work, we choose to highlight only the two most important aspects:

- The tool can monitor any property written in DATEs which is at least as expressive as timed automata with stopwatches — although when using the tool for hard time constraints, one must keep in mind that in Java one cannot guarantee exact timing of events (without going for an underlying real-time virtual machine, which we prefer to avoid).
- Properties can be verified for each tuple of objects. Thus, a monitor is (dynamically) generated for every tuple which is active in the system being verified. Each instance of the monitor, which is an automaton, can

also communicate via channels and global variables with other similar automata and also with a global automaton, making the logic very expressive.

2) Time and memory overhead estimation: LARVA can provide a compiler that accepts Lustre code as input which can be used for monitoring. Assuming that the LARVA compiler is correct, no extra overheads are introduced over and above the Lustre time and memory requirements. Thus, for the subset of properties expressed in Lustre and translated into DATEs, the tool can calculate an upper-bound of the resources required for monitoring. This is a significant advantage in security-critical systems. Admittedly, there is a limitation as to the accuracy of the time required for the monitor execution imposed by the underlying operating system and the Java thread scheduling system. However, in security-critical systems one would assume that the operating system gives priority to the system being executed and that the user opts for Java Real-Time. If this is the case the accuracy limitation should not be a problem. The full account of this work is soon to be submitted for publication [5].

3) Composing features together: The tool also enables specification of properties using Lustre and duration calculus properties on Java code. By first translating the Lustre or the duration calculus formulae into DATEs, the script with the translated properties can be extended so as to be applied *for each* instance of a particular object rather then just on the global system.

III. OBSERVER-MATTERS PROPERTIES

The satisfaction of real-time properties are sensitive to slowing down or speeding up of the underlying system. For example, if a system is transferred to run on a faster machine, some properties may be violated simply due to the system running faster e.g. *no more than 1000 accesses to the databases may occur every second*. Similarly, other bugs may appear when the system is slowed down e.g. *an acknowledgement must be sent no later than 0.1s after receiving a request*. In runtime verification, slowing down occurs when adding a monitor, while speeding up may also occur if the runtime monitors are removed at some stage.

A fragment of the duration calculus [6] called counterexample traces [4] has been identified, for which, as long as the events of the underlying system do not change their order (but the intervals between them become longer or shorter) the validity of the properties remains unchanged [7]. These checks have been implemented in LARVA, thus providing guarantees on real-time properties.

Through an analysis of the specification, LARVA tries to deduce whether adding monitors to the system, thus slowing it down, will have an effect on the validity of the specification. The tool may deduce both truth and falsity preservation under slowing down. For example, consider the property *"no more than three bad logins are allowed in ten minutes"*. No matter how much a system is slowed down by monitoring, there is no possibility that such a property can be violated if it is satisfied in the non-monitored version. In other circumstances, the monitors might be removed from the system and we would like to guarantee that the properties which were respected will remain true: *truth preservation* under *time interval-compression*. Through a syntactic analysis of the property, LARVA can provide such guarantees.

Furthermore, LARVA provides a translation from the investigated subset of the duration calculus into DATEs, which preserves these guarantees.

As a case study of using the duration calculus for runtime monitoring in a real-life scenario, we have used a network intrusion detection system. A number of properties are expressed as duration formulae or counterexample traces, and used to detect possible malicious activities on a network connection. Each of these properties is *stretch truth-preserving*, i.e. if the property holds on a system, it will also hold on a slowed-down version of the system. This fact assures us that inserting monitors in the system, will not cause a violation of any of the monitored properties — no false negatives will occur. Typical properties verified are:

1) **Connection initiation** It is desirable to disable any incoming TCP packets which do not belong to connections initiated by the host machine being monitored. The initialisation of a TCP connection requires a complete three-way handshake: first a synchronization packet from the client, then a synchronization and acknowledgement packet from the server and another acknowledgement from the client. If the host machine receives a synchronization packet without having sent one beforehand, then an outsider is trying to open a connection.
2) **Redirection of messages**
   In the case of a machine with a routing table, a lot of ICMP redirect messages can cause the system to slow down. Therefore, if a number of ICMP redirect messages are received in a relatively short time interval, this may be considered as a threat to the system. The property, which disallows three redirect messages with less than two time units between subsequent messages.
3) **Connection failure retries**
   A denial-of-service attack can be carried out by initiating an excessive number of connection initialisations to a server and then leaving the handshake incomplete. The server will have to wait for each of these initialisations to timeout. Sometimes these timeouts can cause serious availability problems for the server because connection requests can be issued at very high speeds. A simple check would be to limit the number of subsequent failed connection retries originating from the same IP address.

The duration formulae used are universally quantified over state variables, exploiting the inherent parametrisation over tuples of objects used in LARVA. In summary, given a property as a counterexample trace, the following steps are required to monitor a Java program to detect any violations of the property: (i) use the tool to automatically convert the counterexample trace into a LARVA script (at this point the tool

examines the properties and outputs the guarantees which can be given); (ii) relate the monitoring events to system events (such as method calls); (iii) if the property is to be monitored for each object of a particular class, modify the LARVA script accordingly; (iv) add any Java code to be invoked in case of a violation detection; (v) compile the script to generate the monitoring system; and (vi) run the Java program with the generated monitoring files in place.

Note that the properties in the intrusion detection system are all slowdown truth preserving, i.e. slowing down the system will not cause the properties to be violated. For example, consider the fast succession of redirect messages: if the frequency of redirect messages does not violate the above property, there is no way by which the property can be violated by slowing down the system. The other cases are similar.

IV. TOOL IMPLEMENTATION ISSUES

A. Monitor Management

- *The internals of a monitor* In practical terms, a monitor is a class with a number of local variables and object references. When monitoring individual objects, a monitor can be thought of as a wrapper around the object (or tuple of objects) being monitored. An indispensable method in a monitor is the *equals* method which enables the system to distinguish a monitor from another. A monitor class also includes the monitoring logic generated from DATEs and some utility functions for display.
- *Creating monitors* A global-level monitor is created as soon as classes are being loaded by the system. On the other hand, a lower-level monitor is created for an object as soon as a relevant event of that object is detected.
- *Maintaining hierarchy* Due to the hierarchical nature of nested automata, each monitor has a reference to its parent, making all the parent's variables available to the child.
- *Loading monitors* Monitors are stored in a hash map whose key is the monitor itself. This approach provides fast retrieval of a monitor when an applicable event causes an update. Note that the retrieval of a monitor highly depends on the *hashCode* and *equals* methods of the monitored object(s).
- *Discarding monitors* It is challenging to decide when a monitor can be discarded. For example, an object which has been serialised and discarded, might be later loaded again and deserialised. Logically, the object is the same one and it should be monitored by its existing monitor. For this reason, it is up to the user to use *accepting* states signalling that the automaton can be discarded once it reaches an accepting state.

B. Real-Time Issues

- *Single master clock* Since using Java Real-Time is not an option in our case, great care was taken to ensure the best attainable accuracy. Using a thread for each clock creates a chaos of non-determinism and possibly a great number of threads which are a considerable overhead to the system. Thus, the solution is to use a single thread for a master clock. Each clock can register with the master clock to be notified when a particular time period elapses. This approach ensures that clock events are always carried out in the correct order, i.e. ordering is not affected by thread scheduling. The only remaining source of non-determinism is the thread scheduling between the master clock and the system's threads. This problem is lessened by giving the master clock thread a high priority. Furthermore, the problem is almost completely avoided if the transition triggered by the clock does not refer to the system time and modifies only the monitor state but not the system state.
- *"Pausing" time* For providing deterministic behaviour with clocks, one would usually like to work under the assumption that time is *paused* while the monitor is taking a transition. This is provided to the user by allowing access to the precise timestamp at which the clock was triggered throughout the transition condition and action.
- *Thread issues* Introducing a thread in a Java system for the master clock might still be dangerous for the system. The solution to this issue is to avoid modifying the system's state in actions triggered by clock events but simply changing monitor values. Monitor values are guarded with a lock such that only one thread can modify the monitor state at any one time. This locking mechanism also makes the tool usable with multi-threaded systems.

C. Inter-Monitor Communication

Although inter-monitor communication might sound complex, it is in fact quite simple at an implementation level. A channel is always global and upon the call of the *send* method, it broadcasts the message to all the monitors using aspect technology themselves, i.e. using the normal event-detecting mechanism used for all system events.

V. CASE STUDY: FINANCIAL TRANSACTIONS

Apart from the network intrusion detection system, LARVA has been applied to a real-life system — a financial system for handling credit card transactions, monitoring a number of interesting properties, including:

1) **Conditions on events and system state** After the execution of certain crucial events the system should guarantee certain security conditions. For example, upon the logging of an event, we must ensure that no credit card numbers are stored. In the case study, this was achieved by detecting a logging event and analysing the string being logged.
2) **Object life cycles** Usually, an entity in a system has a number of states through which it can go during its lifetime. In our financial system, transactions go through a series of states which must be traversed in a particular order and no state should be left out for a transaction to be successfully completed.

TABLE I
EXPRESSIVITY FEATURES OF VARIOUS TOOLS.

Tool	LARVA	ConSpec	Java-MOP	Java-MaC	Hawk	Lola
Scope	*Sess.*[a]*/Obj.*[b]	✓	*Sess./Obj.*	*Sess.*	*Sess.*	*Sess.*
Exceptions	✓	✓	×	×	×	×
Temporal Logics	×	×	✓	×	✓	✓
Real-Time	✓	×	×	✓[c]	✓[d]	×
Mobile Application Policies	×	✓	×	×	×	×
Invariants	✓	×	✓	✓	×	×
Feedback	✓	*Stop.*[e]	✓	✓	×	×
Conditions	✓	✓	✓	✓	×	×
Numerical Queries	×	×	×	×	×	✓

[a] *Sess* stands for a session scope (i.e. a run of the program).
[b] *Obj* stands for an object scope (i.e. a monitor for each object).
[c] Restricted (cannot trigger clock events).
[d] Can be extended to support real-time.
[e] Can only cause the system to halt completely.

3) **Invariants** A number of attributes of an object should persistently adhere to particular conditions. For example, during the processing of a transaction a number of details in the transaction object cannot be changed. Consider the amount specified on a transaction: it is not desirable that at any state during the communication with the bank system, the transaction amount is doubled.
4) **Counting the number of events** Certain events are bounded on their number of occurrences. The financial system under investigation requires that failed transactions are retried the correct number of times.
5) **Real-time** It is often the case that what is expected is not only occurrence of a certain event, but when such an event occurs or its duration. For example, a user should be given a response within one minute of a given request.

For this case study, an early version of LARVA was used, and helped identify extensions required to address challenges faced when monitoring real-life systems.

VI. SIMILAR TOOLS

LARVA expressivity and performance was compared to a number of other runtime verification tools including Conspec [8], Java-MOP [9], Java-MAC [10], Hawk [11], and Lola [12]. To assess expressivity, we have looked at a number of relevant classes of properties and showed how the tools compare. The results are summarised in Table I. LARVA supports scoping, exceptions, real-time, invariants, feedback and conditions both on event parameters and on system state. However, we do not have explicit support for temporal logics, numerical queries (statistics), and mobile application policies. In the case of resource consumption, we developed a benchmark with a number of typical real-time properties and compared the memory and temporal requirements. When compared to a tool with the same level of expressivity — like Java-MOP — LARVA performed well in terms of resources consumed. The complete analysis can be found in [1].

VII. CONCLUSION

Motivated by the need of better expressivity and guaranteed monitoring of real-time properties, we have developed LARVA. Using a number of translations, we offer the choice to use an appropriate notation. For particular logics, we have developed guarantees regarding the overheads and the impact of monitoring on properties. The case studies and the outcome of the comparison with other tools have shown LARVA to be a promising tool. We believe that the tool and its surrounding theory provide a better platform for the challenging monitoring of real-time properties.

REFERENCES

[1] C. Colombo, G. J. Pace, and G. Schneider, "Dynamic Event-Based Runtime Monitoring of Real-Time and Contextual Properties," in *FMICS'08*. To appear in LNCS, 2008.
[2] P. Pandya, "Specifying and Deciding Quantified Discrete-time Duration Calculus formulae using DCVALID," in *Proc. of RTTOOLS'01*, 2001.
[3] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous dataflow programming language Lustre," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1305–1320, 1991.
[4] J. Hoenicke, "Combination of Processes, Data, and Time," Ph.D. dissertation, University of Oldenburg, July 2006.
[5] C. Colombo, G. J. Pace, and G. Schneider, "Resource-bounded runtime verification of java programs with real-time properties," 2009, working draft.
[6] Z. ChaoChen, C.A.R. Hoare, and A.P. Ravn, "A calculus of durations," *Information Processing Letters*, vol. 40, no. 5, pp. 269–276, 1991.
[7] C. Colombo, G. J. Pace, and G. Schneider, "Safe runtime verification of java programs with real-time properties," in *FORMATS'09*, 2009, to appear in LNCS.
[8] I. Aktug and K. Naliuka, "Conspec: A formal language for policy specification," in *FLACOS '07*, October 2007, pp. 107–109.
[9] F. Chen and G. Roşu, "Java-MOP: A monitoring oriented programming environment for Java," in *TACAS'05*, ser. LNCS, vol. 3440. Springer-Verlag, 2005, pp. 546–550.
[10] I. Lee, S. Kannan, M. Kim, O. Sokolsky, and M. Viswanathan, "Runtime assurance based on formal specifications," in *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, 1999.
[11] M. d'Amorim and K. Havelund, "Event-based runtime verification of java programs," *SIGSOFT Software Engineering Notes*, vol. 30, no. 4, pp. 1–7, 2005.
[12] B. D'Angelo, S. Sankaranarayanan, C. Sánchez, W. Robinson, B. Finkbeiner, H. B. Sipma, S. Mehrotra, and Z. Manna, "Lola: Runtime monitoring of synchronous systems," in *TIME'05*. IEEE Computer Society Press, June 2005, pp. 166–174.