A Practical Approach to Runtime Verification of Real-Time Properties for Java Programs

Christian Colombo
Dept. of Computer Science
University of Malta
ccol002@um.edu.mt

Gordon J. Pace
Dept. of Computer Science
University of Malta
gordon.pace@um.edu.mt

Gerardo Schneider
Dept. of Informatics
University of Oslo, Norway
gerardo@ifi.uio.no

Abstract

Given the intractability of exhaustively verifying software, the use of runtime-verification to verify single execution paths at runtime, is becoming increasingly popular in industrial settings. In this paper we present dynamic communicating automata with timers and events to describe properties of systems, implemented in LARVA*, an event-based runtime verification tool for monitoring temporal and contextual properties of Java programs. We give the mathematical framework behind* LARVA *and show how real time logics can be translated into* LARVA *providing additional benefits to the runtime monitoring framework. These benefits include guarantees on the memory upperbound required for the monitoring system and guarantees on the effect of varying the execution speed of the system with regards to real-time properties.*

Index Terms

runtime verification, real-time properties, duration calculus

1. Introduction

As computer systems become increasingly present in all the aspects of our lives, be it in avionics, medical equipment, on-line billing systems, etc, it is becoming increasingly important to provide reliable and robust software. Faults in security- and safety-critical systems can be financially costly or even cause the loss of human lives.

Until recently, testing was the main, if not the only tool to improve software reliability. Despite the effectiveness of proper testing, it is extremely difficult to test huge software products in a sufficiently thorough manner so as to ensure their correctness due to lack of coverage. Since a system does not work in a vacuum, but in an environment which can be difficult to predict and practically impossible to simulate all possible variations, exhaustively or even effective testing with high coverage a complex system is practically impossible.

Another approach to provide safe and secure software, which has been gaining ground since the 90s, is that of model checking, in which one verifies all execution traces which the system can possibly run into. However, model checking fails to scale up easily on large systems, and is thus usually impractical without strong abstraction techniques using which the model verified and the system may be far removed.

Runtime verification has built upon model checking techniques to verify the correctness of a property over a system, but restricting the coverage of the verification to a single path at a time, just as in testing. However, unlike testing, the verification along the execution path is performed at runtime, ensuring that the system is never allowed to follow undesirable paths. In this way one can guarantee that whatever the system environment and input, the behaviour is still correct. In the case of a property violation, the verifying system can either raise an alarm or take some action to correct the state of the verified system. The underlying assumption of runtime verification, is that arresting the system's execution upon discovery of a problem is sufficient to ensure the correctness of the system. If prefixes of such execution paths necessarily lead to problems, then it is up to the designer or developer to identify these undesirable prefixes which may then be stopped using runtime verification.

In order to provide a description of the specification, one requires a precise notation appropriate to describe typical properties in a clear and succinct manner. A number of logics have been developed some of which are more appropriate in particular domains. It is also

important for the logic to be automatically transformed into monitoring code, to ensure that what you write is what you verify.
One also needs to consider the problem that upon injecting monitoring code, the environment of the target system is also being effectively changed. The overhead of the monitoring code effectively slows down the system, leading to different sampling or reaction time in concurrent and real-time systems. Increased memory overheads are another issue one has to tackle. In this paper, we outline how we tackle the issues in two different ways (i) by guaranteeing an upperbound on time and memory resource requirements for runtime verification; and (ii) by characterising real-time properties which would not cause new violations upon speeding up (or slowing down, depending on whether one looks at adding or removing a monitor) the verified system. This guarantees that one would not introduce errors while eliminating others.

2. LARVA

The expressivity of the logic in which to express properties is crucial — on one hand, the logic should be expressive enough to be able to handle certain features in a practical setting, such as temporal aspects and contextual aspects, but it should also enable the automatic generation of effective and efficient monitors. In this section, we briefly present a theory of communicating automata with events and timers used to express properties, and the construction of monitors from such properties — the basis of our tool LARVA. A more complete presentation can be found in [3].

2.1. Dynamic Automata with Events and Timers (DATEs)

The underlying logic we will use to define the specification properties of the system will be based on communicating symbolic automata with timers, whose transitions are triggered by events — referred to as *dynamic automata with events and timers* (DATEs). Events are built as a combination of visible system actions (such as method calls or exception handling), timer events and channel synchronisation (through which different automata may synchronise).
Definition 2.1: Given a set *systemevent* of events which are generated by the underlying system and may be captured by the runtime monitors, a set of timer variables *timer*, and a set of *channels*, a composite *event* made up of system events, channel synchronisation, a timeout on a timer, a choice between two composite events (written $e_1 + e_2$) or the complement of a composite event (written $\overline{e}$), is syntactically defined as follows:

$$\begin{array}{rcl} \textit{event} & ::= & \textit{systemevent} \mid \textit{channel}? \mid \textit{timer} \ @ \ \delta \\ & \mid & \textit{event} + \textit{event} \mid \overline{\textit{event}} \end{array}$$

We say that a basic event x (which can be a system event, a channel synchronisation or a timeout event) will fire a composite event expression e (written $x \vDash e$) if either (i) x matches exactly event e; or (ii) $e = e_1 + e_2$ and either $x \vDash e_1$ or $x \vDash e_2$; or (iii) x is a system event and $e = \overline{e_1}$, and $x \nvDash e_1$.
The notion of firing of events can be extended to work on sets of events. Given a set of basic events X, a composite event e will fire (written $X \vDash e$) if either (i) e is a basic event expression, and for some event $x \in X$, $x \vDash e$; or (ii) $e = e_1 + e_2$ and either $X \vDash e_1$ or $X \vDash e_2$; or (iii) X contains at least one system event and $e = \overline{e_1}$, and for all $x \in X$, $x \nvDash e_1$.
The semantics of the complement of an event is constrained to fire when at least one system event fires, so as to avoid triggering whenever a timer event or channel communication happens, thus making such events to necessarily depend on the underlying system. This constraint can be relaxed without effecting the results in this paper.
Since we require real-time properties, we will introduce timers (ranging over non-negative real numbers), whose running may be paused or reset. The configuration of a finite set of timers determines the value and state of these timers.
Definition 2.2: The *configuration of the system timers* (written $\mathcal{CT}$) is a function from timers to (i) the value time recorded on the timer; and (ii) the state of the timer (running or paused). *Timer resets*, *pauses* and *resumes* are functions from a timer's configuration to another — changing only the value of one timer to zero (in the case of a reset), or the state of one timer (in the case of pause or resume). A *timer action* (written $\mathcal{TA}$) is the composition of a finite number of resets, pauses and resumes.
Based on events and timers, we define *symbolic timed-automata* — similar to integration automata [1], but more expressive than Alur and Dill's Timed Automata [2] (since we enable reset, pause and resume actions on timers). Unlike integration automata, the automata we introduce have access to read and modify the underlying system state. In practice this can be used to access and modify variables making the transitions more symbolic and enumerative in nature.
Definition 2.3: A *symbolic timed-automaton* running over a system with state of type Θ is a quadruple $\langle Q,\ q_0,\ \rightarrow,\ B \rangle$ with set of states Q, initial state

$q_0 \in Q$, transition relation $\rightarrow$, and bad states $B \subseteq Q$. Transitions will be labelled by (i) an event expression which triggers them; (ii) a condition on the system state and timer configuration which will enable the transition to be taken; (iii) a timer action to perform when taking the transition; (iv) a set of channels upon which to signal an event; and (v) code which may change the state of the underlying system:

$$\langle Q \times \mathit{event} \times (\Theta \times \mathcal{CT} \rightarrow \mathrm{Bool}) \times \mathcal{TA} \times 2^{\mathit{channel}} \times (\Theta \rightarrow \Theta) \times Q \rangle$$

We will assume that a total ordering $<$ exists on the transitions to ensure determinism.

The behaviour of an automaton M upon receiving a set of events consists of (i) choosing the highest priority transition which fires with the set of events and whose condition is satisfied; (ii) performing the transition (possibly triggering a new set of events); and (iii) repeating until no further events are generated, upon which the automaton waits for a system or timeout event.

The notions of symbolic timed-automata can be lifted to work on dynamic networks of symbolic timed-automata, in which we enable the creation of new automata during execution in a structured manner — referred to as Dynamic Automata with Events and Timers (DATE) in the rest of the paper.

Definition 2.4: A DATE $\mathcal{M}$ is a pair $(\bar{M}_0, \nu)$ consisting of (i) an initial set of automata $\bar{M}_0$; and (ii) a set of automaton constructors ν of the form:

$$\mathit{event} \times (\Theta \times \mathcal{CT} \rightarrow \mathrm{Bool}) \times (\Theta \times \mathcal{CT} \rightarrow \mathrm{Automaton})$$

Each triple $(e, c, a) \in \nu$ triggers upon the detection of event e, with the state and timer configurations satisfying condition c, and creating an automaton using function a.

Finally, the events created by the transition can themselves trigger new transitions. Upon receiving a set of system actions, a DATE performs a *full-step* by, not only triggering transitions, but also propagating new events along in the DATE until no more transitions occur. A full definition is given in [3].

Example 2.1: Consider a system where one needs to monitor the number of successive bad logins and the activity of a logged in user. By having access to *badlogin*, *goodlogin* and *interact* events, one can keep a successive bad-login counter and a clock to measure the time a user is inactive. Fig. 1(a) shows the property that allows for no more than two successive bad logins and 30 minutes of inactivity when logged in, expressed as a DATE. Upon the third bad login or 30 minutes of inactivity, the system reverts to a bad state. In the figure, transitions are labelled with events, conditions and actions, separated by a backslash. It is assumed that the bad login counter is initialised to zero.

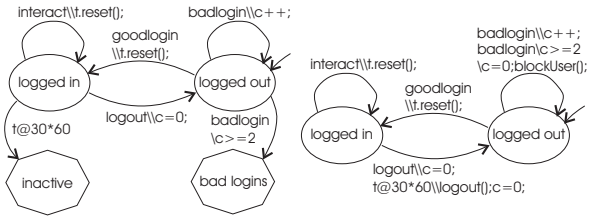


Figure 1. (a) An automaton monitoring the bad logins occurring in a system; (b) The same automaton with recovery actions.

Fig. 1(b) shows how actions can be used to remedy the situation when possible, instead of going to a bad state. For example, after too many bad logins, one can block the user from logging in for a period of time, and upon 30 minutes of inactivity, the user may be forced to logout.

3. LARVA by Example

Dynamic automata with events and timers can be used to express properties, and can then be directly and automatically implemented as runtime-monitors for an underlying system. We will present the system LARVA which embodies the implementation of this architecture.

System Events. As the underlying system events LARVA uses method calls, invocation of exception handlers, exception throws and object initialisations.

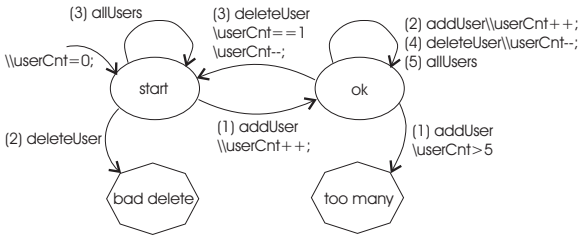
Actions. Actions which are performed on the system state upon each transition are essentially programs which can access code and data from the original program. Note that the model we have used performs the action without triggering any other transition directly. In LARVA this is emulated by ensuring that system events (eg. method calls) are masked from the automaton triggers when called as actions. Furthermore, as shown in the example from the previous section, unless mitigating a problem which arose, it is usually sufficient to constrain actions to access only data local to the automaton. For this purpose, LARVA provides the means to have code local to an automaton.

Dynamic triggers. Dynamic triggers are used in LARVA to enable multiple instances of a property. The property shown in Fig. 2 must be replicated for each bank branch object , in order to make it useful. For this purpose, LARVA enables properties to be replicated for multiple instances of an object — written `foreach object { property }`. Upon capturing events in the property, the system checks whether

it is a new object (using object equality, or a user provided mechanism) and if so creates a new automaton.
Context information. Various properties use nesting of the replicating mechanism — each bank client may have a number of accounts, upon which a number of transactions may take place. Properties about transactions must thus be created for each and every transaction created, but each must have access to its context — the account and client it belongs to. Giving replicated properties access to these inherited values, enables concise and clear properties to be expressed.
Invariants. Furthermore, various objects in the system are expected to satisfy invariants — once set, the ID of a transaction may not change throughout its lifetime. To enable this, LARVA also enables such properties to be expressed, and which are checked upon the arrival of each event. Internally, this is done by creating an implicit transition from each state which sends the automaton to a bad state should the condition not be satisfied.
Real-time. LARVA provides a clock/stopwatch construct that can trigger events after particular time intervals, implemented as Java threads using *wait* operations. The main drawback of this approach is that it may not be totally accurate due to the Java thread scheduling mechanism.
LARVA uses aspect-oriented programming techniques [7] to capture events. Upon running the monitored system, the underlying automata are created and initialised. Through the use of aspect-oriented programming techniques, whenever an event is captured, control is passed back onto the DATE structure, which performs a full-step, performing any timer actions and new timer events scheduled as necessary before returning control to the system to proceed. If the system reaches a bad state in any of the properties, appropriate action is taken to terminate or remedy the situation as specified by the user.
Example 3.1: To illustrate the use of LARVA, consider the monitoring of a simplified banking system, in which we would want to monitor that there should never be more than five users in the bank and that a deletion does not occur when there are no users. By identifying the system events, corresponding to the method calls in the target system.

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.

```
FOREACH (User u) {
  ...
  FOREACH (Account a) {
    INVARIANTS
    { String accID = a.getID(); }
```




```
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()]
   }
 }
}
```

Figure 2. The automaton and LARVA code of Example 3.1.

```
    PROPERTY
    { ... }
  }
}
```

4. Case Study

During its development, LARVA was used on an real-life system handling credit card transactions. The complexity of this system lies not only in the size of the underlying code, which although not exceptionally large, has over 26,000 lines of code, but also in the strong security implications and communication required among various components (including third party systems, such as banks). The system is designed to hold sensitive information of thousands of people — a single leak of sensitive information could undermine the confidence of the users in the system, leading to drastic financial losses. Furthermore, the system has real-time issues and is required to be able to handle over 1000 transactions per minute.
The system is composed of two parts, one handling the transactions and their database and the other handles

the communication to the respective bank or entity which is involved in the transaction. These will be referred to as the *transaction handling system* and the *processor communication system* respectively. The whole system will be referred to as the *transaction system*.

A number of different classes of properties, as described below, were verified at runtime using LARVA on the system.

Logging of credit card numbers. During the development of the original transaction system, credit card numbers were logged for testing purposes. This is however, not in line with standard practice of secure handling of credit card numbers. These logging instances were manually removed from the code. However, to ensure that no instances escaped the developers' attention, a simple verification check to ensure that no data resembling a credit card number is ever logged.

Transaction execution. Transactions are processed by going through a number of stages, including authorisation, communication with the user interface, insertion of the transaction in the database and communication with the commercial entity involved in the transaction, the stages taken depend on which classification the transaction falls under. Designing properties to ensure that during its lifetime, all transaction go through the proper stages was straightforward, especially since the automata-based property language we use corresponds closely to the concept of stages, or modes in which a transaction resides.

Authorisation transactions. Authorisation transactions have to be checked to ensure that all the stages are processed in the correct order, keeping certain values unchanged — for instance, one must make sure that the ID of the transaction is never accidentally changed. Furthermore, other checks such as ensuring that transaction amounts are not changed after being set were also necessary.

Backlog. A particular feature of the system under scrutiny, is that if communication with a bank or a commercial entity fails, the request is retried a number of times after a given delay. This process is called backlogging. The transaction handling system with backlogs can become rather complex — and properties were identified to ensure that the backlog process is performed for the expected number of times or until the transaction is approved. Timeout mechanisms were used to check for failures and retries within a given interval of time, giving warnings and errors if the property is not satisfied.

Given the nature of the system, with different components and transactions communicating and synchronising their behaviour, it was difficult to measure the overhead of the monitoring system for the case study. The case study, however, was essential to identify features necessary for the use of runtime verification on real-life case studies. The need for context-information and invariants arose directly from this experience.

5. Ongoing Work

We are currently exploring various avenues of extending our initial work on LARVA. Three strands we are currently exploring are briefly outlined below:

Supported Logics: Duration calculus [10] is a highly expressive dense-time logic, based on intervals rather than points in time. For runtime verification this has a significant advantage that there is no distinction between past and future. For this reason and its sound mathematical background, duration calculus is ideal to represent certain real-time software properties. Duration calculus is too expressive to be fully implemented, but a useful subset providing enough expressive power for commonly known practical applications has been identified in [8], [9]. This subset can be converted into phase event automata [6] and we are currently working on the conversion of phase event automata into LARVA.

Memory Upperbound Guarantees: If a security critical application has limited resources, for instance in embedded systems, runtime verification has to provide guarantees on the overhead induced through the monitoring. The synchronous language Lustre [5] designed specifically for reactive systems, provides an excellent constrained intermediate language which enables calculation of the memory and execution time required at compile time. Lustre programs are symbolic automata which can be easily converted into LARVA. Thus, properties written in Lustre, can be used to monitor Java programs using LARVA, guaranteeing memory and execution time upperbounds. Furthermore, Gonnord *et al.* [4] have recently shown how QDDC (a subset of duration calculus) can be converted into Lustre, providing an more abstract notation for property specification which still guarantees bounded overheads.

Truth of Properties under System Retiming: Since adding or removing a monitor changes the speed of the underlying system, we are exploring the characterisation of real-time properties which preserve correctness despite slowing down (or speeding up) the system. We have identified subsets of duration calculus which satisfies these desirable properties. Slowdown truth preserving properties remain true under a slowing down of the inputs — adding a monitor to check such a property guarantees that any problems identified by the monitor were already there in the original system, in other words, no new bugs may be introduced by adding

a monitor. Analogously, with speedup truth preserving properties, we know that it is not the monitor slowing down the system which is circumventing bugs. This characterisation of real-time properties gives important information enabling smarter runtime-verification approaches.

6. Conclusions

Runtime verification has been widely used in various different contexts and for a wide variety of systems. The need for a sufficiently expressive logic to be able to specify the system properties succinctly and clearly is essential for confidence in the overall monitoring process. In this paper, we have introduced dynamic automata with timers and events (DATEs) to describe properties of systems which need to be checked for different instances of a class. We have also presented LARVA, a runtime verification implementation of this logic. The combination of timers with dynamic automata enables the straightforward expression of various properties, as illustrated in the use of LARVA for the runtime monitoring of a real-life transaction system.

Using different subsets of duration calculus we have shown how we can exploit various advantages. Starting from the QDDC subset, one can go through Lustre to LARVA guaranteeing memory and exeuction time upperbounds for the monitoring system. Starting from the subset of implementable duration calculus, one can translate down into LARVA by going through phase event automata. In both cases, one can analyse the properties to ensure whether they are truth preserving when the system is slowed down (or sped up). This framework of theory and implemented translations should help to make more practical the monitoring of challenging properties such as real-time and memory-restricted properties.

We see LARVA useful in bridging the implementation with requirements — in a software development environment, the requirements of the software can be expressed in a formal notation very early in the development life cycle. If these are written using LARVA during the actual implementation, the requirements can be directly related to concrete system events such as method calls and exception throws. With little extra overhead, the monitoring of system requirements comes with no additional human intervention.

References

[1] A. Bouajjani, Y. Lakhnech, and R. Robbana. From duration calculus to linear hybrid automata. In P. Wolper, editor, *Proceedings of the 7th International Conference On Computer Aided Verification*, volume 939, pages 196–210, Liege, Belgium, 1995. Springer Verlag.

[2] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[3] C. Colombo, G. J. Pace, and G. Schneider. Dynamic event-based runtime monitoring of real-time and contextual properties. In *13th International Workshop on Formal Methods for Industrial Critical Systems (FMICS 2008)*, Italy, 2008. To be published by Springer Verlag in Lecture Notes in Computer Science.

[4] L. Gonnord, N. Halbwachs, and P. Raymond. From discrete duration calculus to symbolic automata. *Electr. Notes Theor. Comput. Sci.*, 153(4):3–18, 2006.

[5] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[6] J. Hoenicke. *Combination of Processes, Data, and Time*. PhD thesis, University of Oldenburg, July 2006.

[7] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP'97—Object-Oriented Programming, 11th European Conference, LNCS 1241*, 1997.

[8] A. P. Ravn. *Design of Embedded Real-Time Computing Systems*. PhD thesis, Technical University of Denmark, October 1995.

[9] M. Schenke and E.-R. Olderog. Transformational design of real-time systems part i: From requirements to program specifications. *Acta Inf.*, 36(1):1–65, 1999.

[10] Z. ChaoChen, C.A.R. Hoare, and A.P. Ravn. A calculus of durations. *Information Processing Letters*, 40(5):269–276, 1991.

Acknowledgements

The research work disclosed in this publication is partially funded by Malta Government Scholarship Scheme grant number ME 367/07/29.