Dynamic Event-Based Runtime Monitoring of Real-Time and Contextual Properties*

Christian Colombo[1], Gordon J. Pace[1], and Gerardo Schneider[2]

[1] Department of Computer Science, University of Malta, Malta.
[2] Department of Informatics – University of Oslo, Oslo, Norway
{ccol002, gordon.pace}@um.edu.mt, 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 popular. Although the use of runtime verification is increasing in industrial settings, various challenges still are to be faced to enable it to spread further. 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. The combination of timers with dynamic automata enables the straightforward expression of various properties, including replication of properties, as illustrated in the use of Larva for the runtime monitoring of a real life case study — an online transaction system for credit card. The features of Larva are also benchmarked and compared to a number of other runtime verification tools, to assess their respective strengths in property expressivity and overheads induced through monitoring.

1 Introduction

As software systems grow bigger and more complex, and as they influence our lives in more frequent and direct ways, the need for their validation similarly grows. Over the past decades, program validation has become an increasingly important and active research area striving towards certified code with the ultimate holy grail of providing a guarantee of the absence of errors from a given system. Though testing and simulation have been the most widely-used techniques in industry, formal methods have started playing a more important role in program validation. Static analysis and model checking techniques have been improved, but still require expertise to apply on large complex systems, and usually fail to scale up to real-life systems. To address the problem of intractability, an alternative approach which has been developed is that of runtime verification, in which the desired properties are only checked at runtime on the active execution path. Properties are written in a formal logic and then transformed into a runtime monitor which is instrumented with the system to be monitored.

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

Subsequently, the runtime monitor observes the system while it is running, and triggers an appropriate response if a system property is violated. The main issue is a trade off between the expressivity of the logic and the overhead created on the monitored system.

Although the overhead induced through the monitors is undoubtedly crucial in certain application areas, indicating that the expressivity of the logic should be constrained so as to ensure effective monitors, the logic should be able to handle certain features to ensure its utility in a practical setting: (1) temporal aspects, including (i) consequentiality — e.g. 'authentication happens before data access', and (ii) real-time — e.g. 'a transaction takes no more than 30 seconds to execute'; (2) Contextual aspects — the possibility of monitoring objects either globally, grouped according to their container or individually, e.g. 'every account (in a given banking system) must belong to a registered user', or 'a bank transfer (corresponding to a given user) may only be performed without any charge if both accounts are registered (in European countries)'; (3) Exceptions — monitoring the exceptional cases in the execution of a program. Although the application domain clearly determines which type of properties one would need to monitor, ideally all the above should be expressible in the logic used for property specification.

In this paper we present dynamic communicating automata with timers and events to handle the above-mentioned aspects when describing properties of systems, and their implementation for runtime-verification in the tool Larva. Larva has been used in a real-life case study, consisting of a Java program which forms part of a software dealing with credit card transactions. Moreover, we compare our tool with a number of state-of-the-art runtime verification tools[3]: Java-MOP, Java-MaC, Hawk, ConSpec, and Lola.

The paper is organised as follows. In the next section we introduce dynamic automata with timers and events (DATEs) as the underlying property specification logic, and how monitors can be constructed directly from such structures, as implemented in the tool Larva. In section 3, we then present the application of Larva on a credit card transaction systems, and in section 4 we compare it to other tools.

2 Event-Based Runtime Monitoring

As argued in the introductory section, the expressivity of the logic in which to express properties is crucial. In this section we 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.

2.1 Dynamic Automata with Events and Timers

The underlying logic we will use to define the specification properties of the system is based on communicating symbolic automata with timers, with event-

[3] We would like to thank Irem Aktug, Johan Linde, Grigore Roşu, Feng Chen, Oleg Sokolsky and César Sanchez for their assistance in benchmarking their tools.

triggered transitions. Events can be visible system actions (such as method calls or exception handling), timer events, channel synchronisation (through which different automata may synchronise) or a combination thereof.

Definition 1. *Given a set* systemevent *of events which are generated by the underlying system and may be captured by the runtime monitor, 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:*

$$\text{event} ::= \text{systemevent} \mid \text{channel}\,? \mid \text{timer}\ @\ \delta \mid \text{event} + \text{event} \mid \overline{\text{event}}$$

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 depend solely on the underlying system, hence increasing compositionality. 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. *The* configuration of 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 (functional) 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 [3] (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.

Properties of a given system will be expressed as communicating timed-automata. These automata will have access to read and modify the state of the underlying system.

Definition 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:*

$$Q \times \text{event} \times (\Theta \times \mathcal{CT} \rightarrow \text{Bool}) \times \mathcal{TA} \times 2^{\text{channel}} \times (\Theta \rightarrow \Theta) \times Q$$

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.

Definition 4. *For a symbolic timed-automaton $M = \langle Q,\ q_0,\ \rightarrow,\ B\rangle$, we say that a set of system scheduled events X, system state $\theta \in \Theta$, timer configuration T and state q (in which M currently resides),* performs a step *to X', θ' and q', with timer update t' (written $(X, \theta, q) \Rightarrow^T_{t'} (X', \theta', q')$) if $q \notin B$ and $(q_1,\ e,\ c,\ t,\ O,\ f,\ q_2)$ be the largest (in terms of $<$) transition in $\rightarrow$ such that: (i) $q = q_1$; (ii) $X \vDash e$; (iii) $c(\theta, T)$, and the following hold: (i) $t' = t$; (ii) $q' = q_2$; (iii) $\theta' = f(\theta)$; (iv) $X' = O$. If no such transition exists, we write $(X, \theta, q) \Rightarrow^T_{id} (\emptyset, \theta, q)$.*
The notion of automata performing a step can be extended over to a vector of automata communicating via broadcast channels. Given a vector of n automata $\bar{M} = \langle M_1, M_2, \dots M_n\rangle$, in states $\bar{q} = \langle q_1, q_2, \dots q_n\rangle$ and with shared timers in state T, we write that $(X, \theta_0, \bar{q}) \Rightarrow^T_{t'} (X', \theta_n, \bar{q'})$ if (i) for each $1 \leq i \leq n$, $(X, \theta_{i-1}, q_i) \Rightarrow^T_{t_i} (X'_i, \theta_i, q'_i)$; (ii) $t' = t_{n-1} \circ t_{n-2} \circ \dots \circ t_1$; (iii) $X' = X'_1 \cup X'_2 \cup \dots \cup X'_n$.

Note that the order of execution is set by the order of the automata, once again to avoid non-determinism. Clearly, as in any programming with side-effects, the management of actions on the transitions must be carefully handled. Also note that the timer actions are accumulated so as to evaluate all conditions with the same initial timestamps.

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 5. *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:*

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

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. The triggered automata in time configuration T, with events X, in system state θ (written $tr(T, X, \theta)$) is defined to be:

$$tr(T, X, \theta) \overset{def}{=} \{a(\theta, T) \mid (e, c, a) \in \nu,\ X \vDash e,\ c(\theta, T)\}.$$

Finally, the events created by the transition can themselves trigger new transitions.

Definition 6. *The configuration of a DATE consists of (i) the state of the timers; (ii) the state of the underlying system; and (iii) the state of the currently running automata — a vector of a state for each automaton in the network.*
A DATE is said to perform a full-step *from configuration $(T, \theta, \bar{q})$ to configuration $(T', \theta', \bar{q'})$, upon receiving a set of system actions X, (written $(T, \theta, \bar{q}) \models\!\!\Rightarrow_X (T, \theta', \bar{q'})$) if for some number n:*

$$(X_0, \theta_0, \bar{q}_0) \Rightarrow^T_{t_1} (X_1, \theta_1, \bar{q}_1) \Rightarrow^T_{t_2} \ldots (X_n, \theta_n, \bar{q}_n) \Rightarrow^T_{t_{n+1}} (\emptyset, \theta_{n+1}, q_{n+1}),$$

where: (i) $\bar{q}_0 = q$, $\theta = \theta_0$ and $\theta' = \theta_{n+1}$; (ii) the final state of the timer is updated according to the timer's accumulated actions: $T' = (t_{n+1} \circ t_n \circ \ldots t_1)(T)$; and (iii) the automata are updated as required by DATE triggers $q' = q_{n+1} \oplus \bigcup_i tr(T, X_i, \theta_i)$.
Such a step is called an accepting full-step*, if no bad states appear in the intermediate state vectors.*

Clearly, not all situations can perform a full-step — even a single automaton may create events on channels which trigger another transition indefinitely. To resolve the problem of livelock, we must ensure that there is no mutual dependancy over the set of automata.

Definition 7. *The* output channels *of an automaton M, written $out(M)$, is the union of all output channels on the transitions in M. Similarly, the* input channels *of M, written $in(M)$, are the channels appearing on the event label of transitions in M. The* dependency relation *between channels for an automaton M, written $dep(M)$ is defined to be $in(M) \times out(M)$.*
A DATE structure $\bar{M}$ is said to be loop-free *if, for any channel c, $(c, c) \notin (\bigcup_i dep(M_i))^*$.*

The following result states that only loop-free automata can perform a full-step.

Proposition 1. *Given a loop-free collection of automata $\bar{M}$ in states $\bar{q}$, set of system events X and system state θ, there exist states $\bar{q'}$, system state θ' and timers T such that $(T, \theta, \bar{q}) \models\!\!\Rightarrow_X (T', \theta', \bar{q'})$.*

Example 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

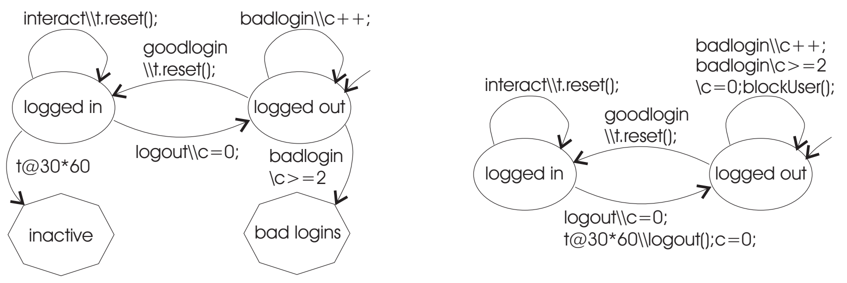


Fig. 1. (a) An automaton monitoring the bad logins occurring in a system; (b) The same automaton with recovery actions.

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.

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 when logged in, the user may be forced to logout.

2.2 Constructing Monitors from DATEs

These automata can be used to express properties, and can then be directly and automatically implemented as runtime-monitors for an underlying system. This transformation has been implemented in the system Larva which embodies the implementation of these properties in an automatic manner.

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. 1 must be replicated for each user attempting to login to the system, 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.

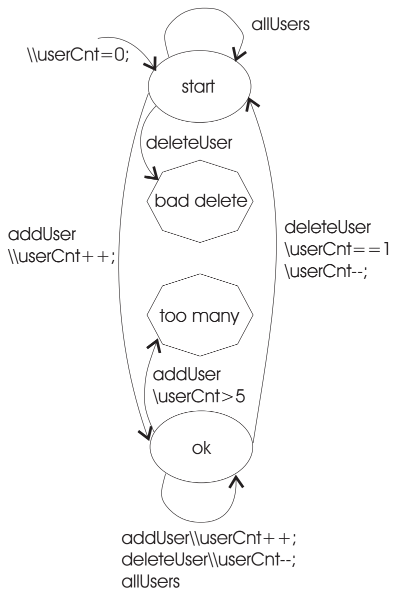
Real-time: Larva provides a clock/stopwatch construct that can trigger events after particular time intervals, implemented as Java threads using *wait* operations. The main drawback of this approach is that it may not be totally accurate due to the Java thread-scheduling mechanism.

Larva uses aspect-oriented programming techniques [9] 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 to the DATE , 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 2. To illustrate the use of Larva, consider the monitoring of a simplified banking system, in which one might want to ensure 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, an automaton is constructed, to specify the properties desired. In Fig. 2 we show the automaton used for monitoring the adding and deleting of users, together with the equivalent Larva code[4].

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. Larva enables the specification of such properties using the `foreach` construct — instances of the properties (automata) appearing within the construct are created for each instance of the class. Furthermore, since when nesting the construct, the properties inside inner replicators have access to the contextual

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



```
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 Example 2.

information (the instance of the outer iterator) under which they appear. In this case, not only the account, but also the user to whom the account belongs will be known without explicitly invoking Java code each time. Invariants of the instances of the class may also be specified directly, for example, to ensure that the account number never changes. The following code illustrates these concepts.

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

    PROPERTY
    { ... }
  }
}
```

3 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 transactions go through the proper stages was straightforward, especially since the automata-based property language used in Larva 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 was a process called backlogging — if communication with a bank or a commercial entity fails, the request is retried a number of times after a given delay. 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.

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.

4 Comparison of Larva with Other Related Tools

In this section we compare Larva with various other runtime-verification tools on a number of criteria, including both in terms of expressivity and overheads induced.

4.1 Related Tools

ConSpec [2] is inspired by PSLang, but restricted to mobile devices with limited resources. A contract is defined for each application and upon installation on a device, the contract is checked against the user's policies. If the application's contract does not comply with the user's policies, the application cannot be installed on the device. In other cases, where the application's contract cannot be definitively checked before installation, a runtime monitor is inlined to the application.

Java-MOP [5] is a monitoring-oriented development environment combining the specification and the implementation of a system. It goes further than runtime verification in that it not only specifies properties to detect violations and raise exceptions, but the violation handling mechanism is itself part of the design of the system's functionality. Hence, the monitoring is not simply an extra check on top of the system but an integral part of the system's design. An appealing feature of Java-MOP is that it can be extended with different logics including FTLTL, PTLTL, ERE and Jass.

The Monitoring and Checking (MaC) architecture [11] intends to bridge the gap between specification and implementation. It has two different specification languages: the Primitive Event Definition Language (PEDL) and Meta Event Definition Language (MEDL) allowing for a clear separation between the definition of the primitive events of a system and the system properties. An implementation of the Monitoring and Checking architecture for Java is Java-MaC [10], which enables automatic instrumentation to have access to the system events. Instrumented programs send an event stream to the event recogniser and to identify higher-level activities, which are in turn processed by the runtime checker to raise an alarm if any of the specified properties are violated.

Hawk [6] is programming-oriented extension of the rule-based Eagle logic. Eagle [4, 8] is a runtime verification tool comprising a rule-based language and an interpreter for it, supporting future and past time logics, interval logics, extended regular expressions, state machines, real-time and data constraints and statistics. It is implemented as a Java library allowing rule parameterisation. Transitions carry a condition (on the input of the state machine but also on the variables which constitute the underlying system) and an action on the variables of the state. Furthermore, the rules expressed in Eagle, can be either maximal or minimal fixpoint semantics, allowing for more flexibility in expressing weak and strong versions of the same operators.

Lola [7] is a synchronous language which allows the user to specify the properties of a program in past and future LTL. The advantage of Lola is that as a synchronous language it guarantees bounded memory to perform online monitoring, but differs from most other synchronous languages in that it is able to refer to future values in a stream. Lola allows the user to collect statistics at runtime and to express numerical queries.

4.2 The Benchmark

A Java program representing a bank processing a number of transactions for a number of users has been developed to experiment with the use of the different systems. The bank system has a database which is used to simulate communication and time delays in the system. When a transaction is executed there are three possible results: success, failure and exception. Upon a failure, the transaction is successively retried for another four times. No retries are performed in case of an exception. Note that the intention of the benchmark case study is primarily to compare property specification and monitoring.

We have identified a number of classes of properties, and concrete examples for the bank processing system, to compare and contrast the use of the different tools.

Scope: The type of scope which can be specified. Types of scope include object (obj.), session (sess.) — one run of the application, multisession — current and previous runs, global — all running applications of a system. This is used to specify on which level the property is verified. For example if the scope is 'object' then the property will be verified for each individual object.

Exceptions: Exception handling and throwing in an application usually represent important events in a system. This aspect represents whether or not the user can express properties which include exception throwing and handling.

Real-time: Real-time refers to whether or not the monitored properties can include real-time. This means that the verification system is able to trigger checks at particular time intervals and compare clock values upon particular system events.

Invariants: We use the term invariants to refer to inbuilt mechanisms in the verification system to monitor the changing of values of variables. The purpose is to be able to verify that certain variables only change when they are supposed to do so.

Feedback: It refers to the capability of the monitoring system to return feedback to the target system. This usually takes the form of a mitigation action in case a violation is found. In other cases this may be limited to stopping the program's execution (denoted by *Stop.* in Table 1).

Conditions This refers to the ability to filter events by applying a condition on the parameters and/or monitoring variables.

Table 1. Expressivity features of various tools.

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

[a] in specification it supports all the mentioned scopes but currently only *session* is supported
[b] restricted (cannot trigger clock events)
[c] can be extended to support real-time
[d] restricted to implementing conditions in violation/validation handling method

Temporal logics: It represents the fact that the tool supports specification written in temporal logics such as LTL.

Mobile application policies: We refer to the ability of defining a security policy which can be partially verified before runtime if the application also specifies its policy. Verifying applications for mobile devices require the monitoring system to be as lightweight as possible.

Numerical queries: This refers to explicit support to expressing numerical queries about statistics of the program being verified.

Table 1 shows which tool have *explicit* support for the aspect being considered. Note that the meaning of the scope object is sometimes referred to as class. In the case of Larva, the same object need not necessarily be the same instance, but be equated through the (optional) use of a user-provided equality method. One advantage of this approach is that when monitoring objects which are serialised and de-serialised, the object before serialisation will still be considered the same as the object afterwards (even though they are not the same instance).

Although with its own limitations, Larva can express a number of interesting classes of properties, not all of them expressible directly in the other tools. Two limitations of Larva are that it cannot support different temporal logics (Hawk, Java-MOP and Lola do have this capability), and it is not suitable for security of mobile devices (in which ConSpec excels).

4.3 Performance of Larva

Five tests have been built for the evaluation of the performance of Larva in terms of overheads. *Test 0* executes a number of transactions but does not violate any of the given properties. Subsequently, *Test 1* violates the invariant property

Table 2. LARVA overheads when with the benchmark example

Test Reference Number	0	1	2	3	4
System without Monitoring					
time(ms)	4	4	6303	3	9
memory(Kb)	23	23	23	70	161
System with Monitoring					
time(ms)	123	120	6395	161	176
memory(Kb)	453	209	160	467	434
System with Monitoring without Clocks					
time(ms)	55	60	n/a	n/a	36
memory(Kb)	432	477	n/a	n/a	378

Table 3. Statistics obtained when trying *Test 0* on variations of the benchmark.

Test Variation	Normal	Time Cons.	Big Obj.	Many Obj.
System without Monitoring				
time(ms)	4	4722	4874	53849
memory(Kb)	23	23	260	2384
System with Monitoring				
time(ms)	123	5321	5458	65153
memory(Kb)	453	418	458	3509

Table 4. The benchmark applied to various tools. (* — no logging, no clocks)

Test Ref. No.	0	1	4	0	1	4
	LARVA*			ConSpec		
time(ms)	27	23	30	7	n/a	n/a
memory(Kb)	91	136	208	54	n/a	n/a
	Java-MOP			Java-MaC		
time(ms)	23	23	52	7	n/a	n/a
memory(Kb)	174	173	312	26	n/a	n/a

by trying to change the transaction amount. *Test 2* violates the property that a retry should occur within two seconds. *Test 3* violates the property the a user cannot have more than five transactions. Finally, *Test 4* violates the property that upon an exception the transaction is not retried.

Table 2 shows statistics for the benchmark when the five tests were run under three different configurations: (i) without monitors; (ii) with monitors for all the properties; (iii) removing the monitors which include clocks with the purpose of investigating the impact of clocks on the monitoring system.

Although the resources required for the program to run without monitors as compared to when it was run with monitors seems huge, the overhead is linear in the size of the automaton used to describe the properties. In fact, increasing the size of the system with regards to the required memory and processing time preserves the complexity as shown in Table 3, in which *Test 0* is used.

The first experiment was to increase the processing time which the system without monitors require to complete the execution. One should notice how the increase from 4722 to 5321 milliseconds is relatively much smaller than that from 4 to 123 milliseconds. In the second experiment the memory required by each object was increased. In this case the total memory used was 458 kilobytes which is very close to the memory initially used for the initial experiment (453 kilobytes).

In order to investigate the real relationship between the size of the monitoring system and the monitored system, the number of monitored objects was increased by a factor of 10. The results obtained substantiate the intuition that the size of the monitor is linear with respect to the number of monitored objects.

It is difficult to have a true and fair comparison of the tools, since they do not all have the same expressive power and not all monitors were implementable on all tools. For instance, none of the other tools handle real-time properties. Another issue is that LARVA issues a report regarding the status of the monitoring system which is not done by other tools. Removing code and properties to enable common ground comparison between the tools, the results obtained are shown in Table 4. Note that Hawk and Lola are not in Table 4 since we did not have access to the tools. The results concerning their expressiveness were based on descriptions of the tools from papers and personal communication.

The most similar tool to LARVA is undoubtedly Java-MOP since it can implement the same properties in a very similar fashion and both LARVA and Java-MOP use AspectJ as the underlying framework. Compared together, the statistics show that there is little difference. ConSpec is restricted to security properties on mobile devices so the extent of the comparison is limited. To give an idea of resources used by ConSpec, we implemented the property which limits the number of users rather than the number of transactions per user. This explains why the time and memory required were much less than LARVA and Java-MOP. Finally, with Java-MaC, it is again difficult to compare the results since none of the properties could be implemented directly. Furthermore, Java-MaC uses a different technology — transmiting the event stream to other applications running simultaneously. These factors explain the difference in the amount of resources used.

5 Conclusions

Runtime verification has been widely used in various different contexts and for widely different systems. Automatically instrumenting code managing verification from properties gives various advantages. However, 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 communicating automata with timers and events (DATE) 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-time transaction system.

LARVA performs well in comparison to state-of-the-art runtime verification tools, and in terms of expressivity it comprises a set of features not presented as a whole in other tools.

So far the main limitation of LARVA is that it does not support the specification using different temporal logics. This is however not a drawback since the underlying automata theory of Larva are highly expressive, and we are currently implementing a translation from LTL and Duration Calculus to DATEs.

Further Work. We consider that Larva to be mature enough to be used in the development phase to monitor programs were performance is not crucial, even if the overheads induced have been shown to be reasonable. Real-time properties are fragile under slowing down (by introducing monitors) or speeding up (by removing them), which makes runtime verification even more challenging. We are currently building a theoretical framework for the analysis of real-time properties to ensure they are invariant up to slowing down (or speeding up) the system. Larva is being extended with this analysis to enable more confidence in the instrumentation and deinstrumentation of real-time properties.

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. I. Aktug and K. Naliuka. Conspec: A formal language for policy specification. In *FLACOS '07*, pages 107–109, Oslo, Norway, October 2007.
3. R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.
4. H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In *VMCAI*, pages 44–57, 2004.
5. F. Chen and G. Roşu. Java-mop: A monitoring oriented programming environment for java. In *TACAS'05*, volume 3440 of *LNCS*, pages 546–550. Springer-Verlag, 2005.
6. M. d'Amorim and K. Havelund. Event-based runtime verification of java programs. *SIGSOFT Softw. Eng. Notes*, 30(4):1–7, 2005.
7. 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*, pages 166–174. IEEE Computer Society Press, June 2005.
8. A. Goldberg and K. Havelund. Automated runtime verification with eagle. In *MSVVEIS*, 2005.
9. 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.
10. M. Kim, M. Viswanathan, S. Kannan, I. Lee, and O. Sokolsky. Java-mac: A runtime assurance approach for java programs. *Formal Methods in System Design*, 24(2):129–155, 2004.
11. 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.