TECHNICAL REPORT

Report No. CS2009-01
Date: December 2009

Resource-Bounded Runtime Verification of Java Programs with Real-Time Properties

Christian Colombo
Gordon J. Pace
Gerardo Schneider

Department of Computer Science
University of Malta
Msida MSD 06
MALTA

Tel: +356-2340 2519
Fax: +356-2132 0539
http://www.cs.um.edu.mt

Resource-Bounded Runtime Verification of Java Programs with Real-Time Properties

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

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

Gerardo Schneider
Department of Applied IT
University of Gothenburg,
Gothenburg, Sweden
(Department of Informatics
University of Oslo, Norway)
gersch@chalmers.se

Abstract: *Given the intractability of exhaustively verifying software, the use of runtime verification, to verify single execution paths at runtime, is becoming increasingly popular. Undoubtedly, the overhead introduced by runtime verification is a concern for system developers planning to introduce this technique in their work. By using Lustre to write security-critical properties, we exploit the language's guarantees on bounded resources. We translate these properties into the existing monitoring framework* LARVA*, making monitoring of programs both easily applicable to Java programs and at the same time guarantee to use bounded-resources. We use a subset of Quantified Discrete-time Duration Calculus (QDDC) as an alternative specification notation for real-time properties because it is translatable into Lustre. Thus, QDDC also enjoys the same guarantees given when using Lustre.*

Resource-Bounded Runtime Verification of Java Programs with Real-Time Properties*

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

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

Gerardo Schneider
Department of Applied IT
University of Gothenburg,
Gothenburg, Sweden
(Department of Informatics
University of Oslo, Norway)
gersch@chalmers.se

Abstract: *Given the intractability of exhaustively verifying software, the use of runtime verification, to verify single execution paths at runtime, is becoming increasingly popular. Undoubtedly, the overhead introduced by runtime verification is a concern for system developers planning to introduce this technique in their work. By using Lustre to write security-critical properties, we exploit the language's guarantees on bounded resources. We translate these properties into the existing monitoring framework* LARVA*, making monitoring of programs both easily applicable to Java programs and at the same time guarantee to use bounded-resources. We use a subset of Quantified Discrete-time Duration Calculus (QDDC) as an alternative specification notation for real-time properties because it is translatable into Lustre. Thus, QDDC also enjoys the same guarantees given when using Lustre.*

*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 Introduction

When monitoring a system, the system's behaviour is indirectly also modified, which may have undesirable effects, especially if the system has to react instantaneously to its environment. Modification is even more undesirable when monitoring real-time properties which are sensitive to delays and overheads. Although it is virtually impossible to monitor a system without introducing overheads, one would like to, at least, quantify the induced overhead load. For this purpose we suggest a specification framework based around the language Lustre, for which required runtime resources can be computed statically.

Synchronous languages, such as Lustre and Esterel, have been designed to be used to implement reactive systems, in which the underlying system has to react instantanously to the inputs — sometimes called *the synchrony hypothesis*. Since this is obviously impossible to implement in practice, these languages have been designed such that the computation required at each step could be quantified statically at compile time.

In this paper, we propose the use of reactive synchronous languages for runtime verification, to enable computation of overheads at compile time. In particular, we present the use of Lustre as a specification language for Larva — a runtime verification tool for Java programs — thus guaranteeing a bound on the resources required for verification for each event the system produces. We further extend the framework so that Lustre can be used to represent some real-time properties through the use of timestamps. Moreover, apart from Lustre, we also use a deterministic fragment of quantified discrete-time duration calculus (QDDC) for which a translation to Lustre exists. This configuration makes our monitoring framework versatile, allowing specification of properties in different formalisms but at the same time with guaranteed overheads. Finally, we use a case-study of a network intrusion detection system to illustrate the practicality of our approach.

2 Background

In this section we will first introduce dynamic automata with timers and events (DATEs), used for the specification of properties. Subsequently, we give a brief overview of the tool Larva which is an embodiment of DATEs with a complementing framework enabling runtime verification of Java programs. We then present quantified discrete-time duration calculus, and Lustre.

2.1 Dynamic Automata with Timers and Events

The underlying logic we will use to define the specification of properties of the system to be monitored will be based on communicating symbolic automata with timers, whose transitions are triggered by events — referred to as *dynamic automata with timers and events* (DATEs). Events are built as a combination of visible system actions (such as method calls or exception handling), timer events and channel synchronisation. For full definitions, refer to [CPS08].

Definition: A *symbolic timed automaton* (*STA*) running over a system with state of type Θ, is a quintuple $\langle Q,\ q_0,\ \rightarrow,\ B,\ A\rangle$ with set of states Q, initial state $q_0 \in Q$, transition relation $\rightarrow$, bad states $B \subseteq Q$, and accepting states $A \subseteq Q$ (with $A \cap B = \emptyset$). Each transition is labelled with: (i) an event expression which triggers the transition; (ii) an enabling condition on the system state and timer configuration; (iii) a timer action performed when the transition is taken; (iv) a set of channels upon which to signal an event; and (v) code which may change the state of the underlying system. We assume a total ordering $<$, giving a priority to transitions, to ensure determinism.

The behaviour of an STA M, upon receiving a set of events, consists of: (i) choosing the enabled transition with the highest priority; (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.

Dynamic Automata with Timers and Events (DATE) are essentially networks of STAs allowing the creation of new automata during execution.

Definition: A DATE $\mathcal{M}$ is a pair $(\overline{M}_0, \nu)$ consisting of: (i) an initial set of STAs $\overline{M}_0$; and (ii) a set of automaton constructors ν.

DATEs are more general than integration automata [AYR95] and timed automata [AD94] since in DATEs it is possible to enable reset, pause and resume actions on timers. Unlike integration automata, our automata can read and modify the underlying system state. In practice, this can be used to access and modify variables making the transitions more symbolic than enumerative in nature. Full definitions can be found in [CPS08].

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(left) 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,

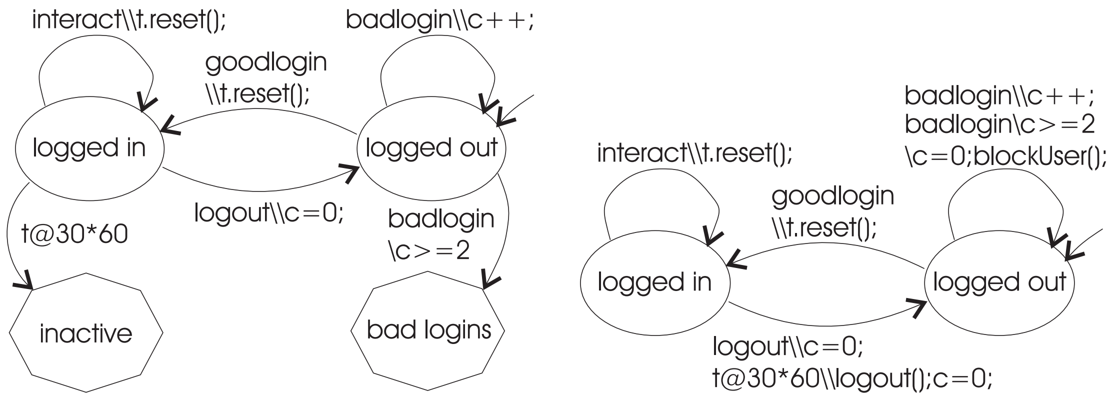

Figure 1: Left: An automaton monitoring the bad logins occurring in a system; Right: The same automaton with recovery actions.

separated by a backslash. It is assumed that the bad login counter is initialised to zero.

Fig. 1(right) 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.

The tool Larva accepts a textual description of a DATE and automatically generates runtime monitors so that the DATE is executed in parallel with an intended Java system. A part of the code corresponding to Fig. 1(left) is shown below:

```
GLOBAL {
  VARIABLES { Clock t; int c = 0; }
  EVENTS {
    interact() = {*.action()}
    t30() = {t@30*60}
    ...
  }
  PROPERTY users {
    STATES {
      BAD { inactive badlogins }
      NORMAL { loggedin }
      STARTING { loggedout }
    }
    TRANSITIONS {
      loggedout -> loggedin  [goodlogin\\t.reset();]
      loggedout -> loggedout [badlogin\\c++;]
```

```
    ...
} } }
```

Upon running the monitored system, the underlying automata are created and initialised. Through the use of aspect-oriented programming techniques [KLM+97], whenever an event is captured, control is passed back onto the DATE structure. Such an event may be a method call, an invocation of exception handler, or the throwing of an exception. Each transition may include an action in the form of Java code. Thus, upon the trigger of a transition, the DATE performs a full-step, after which control is returned back 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.

2.2 QDDC

A number of relevant real-time properties can be represented using an interval-based logic such as Quantified Discrete-Time Duration Calculus QDDC [Pan01]. QDDC is a subset of duration calculus [ZCA91, HC97] based on a discrete-time model. In this work, we are particularly interested in a deterministic fragment of QDDC [GHR06] which can be expressed as Lustre acceptors.

The bases of QDDC are the *state variables* which vary over discrete time. Let $Pvar = \{p, q, \ldots\}$ be the set of propositional variables; $Prop = \{P, Q, \ldots\}$ the set of propositions; and F, F_1, F_2 range over QDDC formulae. The constantly true and false state variables are denoted using *1* and *0* respectively. The set of propositions $Prop$ is inductively defined on state variables with the usual propositional logic operators: $Prop ::= 0 \mid 1 \mid p \mid Prop \wedge Prop \mid \neg Prop$.

Given a path σ (a function from time to basic proposition evaluations), one can define a satisfaction relation $\sigma_t \models P$, meaning that P holds at time t in path σ.

The deterministic QDDC fragment we use in this paper is defined inductively as follows:

$$
\begin{aligned}
G &::= begin(P) \mid [[P]] \mid \eta \leq c \mid \Sigma P \leq c \mid age(P) \leq c && (1)\\
&\quad \mid\; G_1 \wedge G_2 \mid G_1 \vee G_2 \\
F &::= G \mid end(P) \mid G \; then \; F \mid F_1 \wedge F_2 \mid \neg F && (2)
\end{aligned}
$$

Note the two-level definition, which is mainly used to restrict the use of the *then* constructor. The intuitive meaning is as follows:

$$
\begin{array}{lcl}
\sigma[b,e] \models begin(P) & \stackrel{\text{def}}{=} & \sigma_b \models P \\
\sigma[b,e] \models \lceil\lceil P \rceil & \stackrel{\text{def}}{=} & b < e \; and \; \forall i, b \leq i < e, \sigma_i \models P \\
\sigma[b,e] \models \eta \leq c & \stackrel{\text{def}}{=} & (e-b) \leq c \\
\sigma[b,e] \models \Sigma P \leq c & \stackrel{\text{def}}{=} & \#\{i \mid b \leq i < e, \sigma_i \models P\} \leq c \\
\sigma[b,e] \models end(P) & \stackrel{\text{def}}{=} & \sigma_e \models P \\
\sigma[b,e] \models F_1 \wedge F_2 & \stackrel{\text{def}}{=} & \sigma[b,e] \models F_1 \; and \; \sigma[b,e] \models F_2 \\
\sigma[b,e] \models F_1 \vee F_2 & \stackrel{\text{def}}{=} & \sigma[b,e] \models F_1 \; or \; \sigma[b,e] \models F_2 \\
\sigma[b,e] \models \neg F & \stackrel{\text{def}}{=} & \sigma[b,e] \not\models F \\
\sigma[b,e] \models G \; then \; F & \stackrel{\text{def}}{=} & \exists m : \mathbb{N} \; \cdot \; b \leq m < e, \sigma_{[b,m]} \\
 & & \models G \wedge \sigma_{[b,m+1]} \not\models G \wedge \; \sigma_{[m+1,e]} \models F \\
\sigma[b,e] \models age(P) \leq c & \stackrel{\text{def}}{=} & e - n \leq c, \quad \text{where:}
\end{array}
$$

$$
n = \begin{cases} max\{i \mid b \leq i \leq e, \sigma_i \models \neg P\} & \text{if } \neg P \text{ occurred in [b,e]} \\ b-1 & \text{otherwise} \end{cases}
$$

Figure 2: Formal semantics of deterministic QDDC.

- $begin(P)$: given a variable P, returns true if the variable is true at the beginning of the interval;
- $\lceil\lceil P \rceil$: returns true if P is true throughout the interval being considered excluding the final state;
- η: returns the length of the interval being considered;
- ΣP: counts the positive occurrences of P (in number of discrete time points in the interval);
- $age(P)$: returns the duration for which a given variable P has been continuously true up to the end of the interval;
- $end(P)$: given a variable P, returns true if the variable is true at the end of the interval;
- G *then* F: it is satisfied if there is a way of splitting the interval into two such that G holds over all prefixes of the first subinterval but no further, and F holds over the second subinterval (it is a deterministic chop operator).

The formal semantics is given in Fig. 2, defining $\sigma[b,e] \models F$, meaning that F holds over the time interval between time b and time e on trace σ.

As an example, let us consider the following QDDC formula: $(age(Danger) < 5) \vee (age(Alarm) > 0)$. This simply states that either the *Danger* signal has been on for less than 5ms or the *Alarm* should have started to sound.

2.3 Lustre

In real-time systems, one usually requires reactivity — instantaneous response to the stimuli of the environment, to ensure that the input sampling occurs fast enough. Synchronous languages are a class of languages specifically designed for programming reactive systems. The synchronous nature of these languages lies in the assumption that the reaction time of the system is considered to be negligible — the system outputs are available before the next inputs become available. One such language is Lustre [CPHP87], for which time and memory requirements of the generated code can be calculated at compile time. This is crucial in critical systems since it enables us to give guarantees on the upper bound of the monitoring overhead.

Lustre programs are defined in terms of nodes, which process streams of values changing over discrete time. Only two time-dependant operators are available in Lustre. Through the use of the `pre` operator, one can access the value of a stream one time-tick in the past, and the `->` operator is used to create a stream behaving like the first parameter at the first time unit, and like the second in the remaining time. A simple Lustre example is shown below:

```
node BadAccess (w,r,i,o:bool) returns (bw,br:bool);
var l:bool;
let
  l = not o and (i or (false->pre(l)));
  bw = w and not(l);
  br = r and not(l);
tel
```

The above piece of code monitors four events: write (w), read (r), login (i) and logout (o). Essentially, it keeps track of whether a user is logged in or not (using the variable l); if a read or write event occurs while the user is logged out, br (or bw respectively) is set to true. Note that no mutual recursion may be used in a Lustre program unless going through a `pre` operator.

Lustre can also be considered as an executable temporal logic, which has motivated the specification of both the system and its properties to be done in the language [HLR92], similar to the concept of *observers* [HLR93]. An observer is a program which checks that the main program keeps to its specification. Hence, the verification problem involves (i) the composition of the two Lustre programs, and (ii) ensuring that the observer never reaches an undesired state.

2.3.1 Monitoring QDDC using Lustre

The conversion from QDDC into Lustre has been given in [GHR06]. Here we give an idea of the process without going into details. For each QDDC formula we can construct a Lustre node which returns true if the formula is satisfied and false if otherwise. Such a node is referred to as an *acceptor*. An acceptor is activated by a special input *begin* which signals the start of the interval being considered. Subsequently, the acceptor returns true for as long as the QDDC formula is satisfied.

To illustrate this construction let us consider an example: Let $begin(p)$ be a QDDC formula which is satisfied if the first state of an interval satisfies p. To mark the start of an interval we use the variable b such that the first occurrence of b is considered as the first state of the interval. The acceptor of $begin(p)$ should return true if at the beginning of the interval (i.e. when b is true) p is true. Thereafter, the acceptor should return true till the end of the interval. For this purpose, we apply a standard node *after* on the conjunction of p and b: $after(b \ and \ p)$, which returns true on the point where its input is true and thereafter. To be more generic, p can in fact be the outcome of another acceptor. Hence, the acceptor of $begin(p)$ may require a number of additional inputs $\mathcal{I}$. Thus, the resulting acceptor, denoted by $\mathcal{A}_{begin(P)}(b, \mathcal{I})$, will be defined by $after(b \ and \ \mathcal{A}_P(\mathcal{I}))$, where $\mathcal{A}_P$ represents the acceptor of P.

The above example of creating the *begin* acceptor shows how acceptors are used by other acceptors. Adopting this approach, we can recursively define acceptors for all QDDC formulas by starting from a number of basic Lustre nodes.

2.3.2 Example of Monitoring using Lustre

Let us consider a mine pump where an alarm must be triggered within five time units after a sensor sends a signal $Danger$. This property can be written in deterministic QDDC as: $age(Danger) < 5 \vee age(Alarm) > 0$

We show next part of the code obtained by using the translation given in [GHR06]:

```
acceptor(b:bool; Danger:bool; Alarm:bool) returns(p:bool);
let
  node_after_1 = (b)?(true):(pre_after_1);
  node_age_1 = (node_after_1 && Danger)?(pre_age_1 + 1):(0);
  node_after_2 = (b)?(true):(pre_after_2);
  node_age_2 = (node_after_2 && Alarm)?(pre_age_2 + 1):(0);
  p = node_age_1 < 5 || node_age_2 > 0;
  pre_age_1 = node_age_1;
  pre_after_1 = node_after_1;
```

```
  pre_age_2 = node_age_2;
  pre_after_2 = node_after_2;
tel
```

The above Lustre code can be translated into LARVA as explained in the next section.

3 Resource-Bounded Monitoring of Duration Formulae using LARVA

Given the computability of resources of a Lustre program at compile time, one can use Lustre to write properties in special Lustre nodes known as *acceptors*. An acceptor observes a number of state variables and returns true or false depending on whether the current state satisfies a given property or not. By faithfully translating Lustre acceptors into LARVA, we can give an upper-bound guarantee of the resources required by that program. Effectively, we would have resource upper-bound guarantee on properties monitored by LARVA on Java programs.

3.1 Going from Lustre to LARVA

A Lustre program can be considered as a group of modules known as nodes. Each node includes one or more assignments of output streams in terms of input streams. These assignments are executed upon system events which generate a new value for each input stream. Using standard Lustre compiling techniques, the nodes in a Lustre program can be automatically merged into one (a process known as *node flattening*). The end result of a complete flattening is a single node with all the assignments in the flattened nodes. Let $\bar{v}$ represent a vector $\langle v_1, v_2, \dots v_n \rangle$ of variables, $\bar{e}$ a vector $\langle e_1, e_2, \dots e_n \rangle$ of expressions, and $\bar{p}$ a vector $\langle p_1, p_2, \dots p_m \rangle$ of variables storing past values of other variables. Then, the above-mentioned assignments can be mathematically expressed as a conjunction of expressions being assigned to variables: $\bigwedge_{1 \leq i \leq n} v_i := e_i(v_1, \dots, v_n, p_1, \dots, p_m)$. Together, the vectors $\bar{v}$ and $\bar{p}$ constitute the memory required by the program. Assuming that the first m variables in $\bar{v}$ require their previous value to be stored in a corresponding variable in $\bar{p}$, the following assignments are required: $\bigwedge_{1 \leq i \leq m} p_i := pre(v_i)$. To resolve the order of execution of the assignments, a topological sort is applied, allowing a computer to execute them sequentially. Then using standard techniques from [HCRP91] one can generate an executable version of the Lustre program as a DATE. The next step is to decide when this transition is to be taken and to connect the Lustre variables to the system

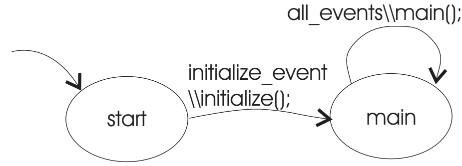

Figure 3: The DATE equivalent to a Lustre program.

events. For this purpose, we basically identify Larva events, upon which the Lustre variables may change their values. These same events will be used on the transition as a trigger. For simplicity, we have so far ignored the fact that some Lustre variables require an initialisation.

The result of the translation process is shown in Fig. 3, where the initialisation code occurs on the first transition (from *start* to *main*) while the code of the flattened node occurs on the other transition (from *main* to *main*).

3.2 Example of Monitoring in Lustre

As an example of a translation from Lustre to Larva, we will consider the code given in Section 2.3. The original code is given on the left hand side while the equivalent list of assignments is shown on the right:

```
l = not o and (i or (false->pre(l)))        l  = e1(o,i,p1)
                                            p1 = pre(l)
bw = w and not(l);                          bw = e2(w,l)
br = r and not(l);                          br = e3(r,l)
```

Note that each statement is represented by an expression in $\bar{e}$ (`e1`, `e2`, `e3`), and that a new variable `p1` has been introduced to store the previous value of l. Also note that the expressions are in the correct sequence because `bw` and `br`, which depend on `l`, are evaluated after the evaluation of `l`. Therefore, the above four lines of code should be placed as an action on the main transition of the resultant DATE. Furthermore, the previous value of `l` requires an initialisation to *false*. Thus, `p1` should be initialised as such on the first transition. On a final note, the events which should be used as triggers would include *login*, *logout*, *read*, and *write*. The resultant Larva script would be as follows:

```
GLOBAL {
  VARIABLES { boolean l, p1; }
  EVENTS {
login(boolean i, boolean o, boolean bw, boolean br) = {*.login()}
```

```
      where { i = true; o = false; bw = false; br = false; }
logout(boolean i, boolean o, boolean bw, boolean br) = {*.logout()}
      where { i = false; o = true; bw = false; br = false; }
    ...
    all(boolean i, boolean o, boolean bw, boolean br)
      = {login | logout | ...}
  }
  PROPERTY access {
    STATES { NORMAL { main } STARTING { start } }
    TRANISITIONS {
      start -> main [init\\p1 = false;]
      main  -> main [all\\l = e1(o,i,p1); bw = e2(w,l);
                          br = e3(r,l);   p1 = pre(l);]
} } }
```

3.3 Real-Time Events

In Lustre, a body of code is executed upon the arrival of new values of variables. However, the arrival of new values need not be synchronised with a regularly beating clock. A substantial class of real-time properties can be runtime-verified through the addition of timestamps to the incoming events and variable values. We extend Lustre nodes to include an additional parameter (a real valued stream *rt_clock*), which will contain the system time at the point when new variable values are generated.

To incorporate this with the work done in [GHR06], one can now easily give definitions for real-time integrals and real-time *age* of expressions. The real-time integral given in the code below returns the total number of time units for which the boolean expression P was true.

```
node rt_integral (rt_clock:time; P:bool; b:bool)
 returns (realtime:time);
let
  realtime = if (strict_after(b) and false -> pre P)
    then (0 -> pre(realtime)) + rt_clock - (0 -> pre(rt_clock))
    else (0 -> pre(realtime));
tel
```

The real-time *age* gives the amount of (real) time during which a proposition has been continuously true, and can similarly be defined as follows:

```
node rt_age (rt_clock:time; P:bool; b:bool) returns (realtime:time);
```

```
var temptime: time;
let
  temptime = if (not (after(b) and (p))) then rt_clock
             else (0->pre(temptime));
  realtime = rt_clock - temptime;
tel
```

4 Case Study

To illustrate the use of QDDC and Lustre we will consider a network intrusion detection system. This consists of monitoring four properties: (i) one written in deterministic QDDC (initiating connections), (ii) one using Lustre (gathering statistics), (iii) one using a construct we define (redirection of messages), and (iv) another using Lustre with multiple nodes (port scan). We also discuss other properties and scenarios.

Initiating Connections. For strict security concerns, one may wish 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 synchronisation packet from the client, then a synchronisation and acknowledgement packet from the server and another acknowledgement from the client. If the host machine receives a synchronisation packet without having sent one beforehand, then an outsider is trying to open a connection.

Monitoring for incoming connections can be easily represented in QDDC as

$$[[\neg \mathit{sendSYN} \wedge \neg \mathit{receiveSYN}]] \ \mathit{then}\ \mathit{begin}(\mathit{receiveSYN}),$$

where the definition of $[[P]]$ is taken to be $[[P] \wedge \mathit{end}(P)$. This formula can be automatically translated into Larva and monitored on Java programs. The only manual intervention required is to connect the variables to Java method calls. Also note that this formula should ideally be monitored for each IP address. In this case, the generated Larva code should be modified to include context and replicate the monitoring automaton for each IP address.

Gathering Statistical Data. By its reactive nature, Lustre code is triggered by inputs from the environment. This makes it ideal to gather statistical data. For standard statistics, the arithmetic operations available in Lustre even suffice to manipulate and extract conclusions while gathering the data itself. Consider the case where we want to limit the number of maximum open connections to one IP address:

```
node max_limit (b:bool; connect:bool) returns (p:bool);
```

```
var count:int;
let
  count = if after(b) then
            if (connect) then 0->pre(count)+1 else 0->pre(count)
          else 0;
  p = if (count > maximum) then false else (true->pre p);
tel
```

Note the usefulness of the idea of using *count*. In fact, in our implementation, this is defined as a separate node. Sometimes it is also useful to count only the number of times when the input becomes true (i.e. rising edges).

Redirect Messages. In the case of a machine with a routing table, a lot of ICMP redirect messages can cause the system to slow down. If this happens in a relatively short time interval, this may be considered as a threat to the system. The property can be easily written using the added construct *bounded* which is violated if the number of occurrences of a proposition exceeds a particular limit within a given period of time. For example, if we want to disallow more than five redirect messages within one thousand milliseconds, we write it as $bounded(redirect, 5, 1000)$, where the acceptor of *bounded* can be defined as follows:

```
node bounded (b:bool; rt_clock:time; redirect:bool; const n:int;
 period:time) returns (p:bool);
var now:bool; count:int; prev_n:time^n;
let
  now = redirect and after(b);
  count = if (now) then (0->pre count)+1 else (0->pre count);
  prev_n = rt_clock | prev_n[0..n-2];
  p = if (count > n and now
      and (rt_clock-(0-> pre(prev_n[n-1])) <= period))
        then false else (true->pre p);
tel
```

Note that the variables *b* and *rt_clock* need not be specified by the user because they depend on the underlying system. Also, we are assuming that each time that *redirect* is true, a redirect message has been received.

Port Scan. A port scan is the discovery of open ports on a system by trying to initialise connections on a range of (usually) well-known ports. To monitor a port scan one should monitor frequent unsuccessful connections from an IP address to different ports. This is not possible to write in QDDC because we need to distinguish among different port numbers. Therefore, in this case we will use two Lustre nodes.

A node will receive a port number and returns true if the port number is different from the last three port numbers received. The second node returns true if three packets with unique ports are received with less than 2000 milliseconds between each two subsequent packets. A final node will be used to connect the two nodes together. The code is as shown below:

```
node uniqueport (receive: bool; port: int) returns (unique:bool);
var port1, port2, port3: int; cnt: int;
let ... tel

node portscan (_rt_clock:int; unique:bool) returns (violated:bool);
var cnt: int; time: int;
let ... tel

node main (_rt_clock:int; receive: bool; port: int)
 returns (violated:bool);
let ... tel
```

Other Properties and Scenarios. The above examples of properties are only intended to give an idea of what can be done using Lustre and QDDC. A more extensive account of the case-study can be found in [Col08]. Other properties that can be written for the above scenario may include, for instance, specialised properties to monitor different aspects of a server, with a higher-level property receiving information from the other monitors. Specialised monitors may be used to keep track of pending, open and active connections, etc. Such specialised monitors can also be used to gather statistics and information from different parts of a system.

For instance, in a scenario with a web server, one may want to gather statistics concerning user access to web pages including the sequence of access to web pages, the most frequently accessed pages, and so on — all valuable information to build user models, improve website linkage, identify the most important pages, and so on. In the case of a file transfer protocol, one may want to gather statistics concerning the actual throughput delivered to the user.

Computing Resource Bounds. Given a Lustre program, we know the exact number of arithmetic operations, assignments and variables. Thus, it is easy to calculate the number of CPU cycles and memory required for the execution of a Lustre program. Obviously, the actual numbers of CPU time and memory depend on the underlying architecture. Considering the above properties which were monitored, the number of operations and variables do not even exceed a hundred. Thus, the overhead incurred is quite small.

5 Conclusions and Related Work

To our knowledge, the only bounded resources guarantee tool for runtime verification is Lola, presented by D'Angelo et al. [DSS+05]. Lola is a synchronous language which allows the user to specify the properties of a program in past and future LTL. Although a synchronous language, Lola differs in that it is able to refer to future values in a stream. This is not possible in other executable synchronous languages (such as Lustre) — Lola is descriptive rather than executable. Lola allows the user to collect statistics at runtime and to express numerical queries. Also, triggers can be defined to generate notifications when a particular boolean expression becomes true, but the tool does not support real-time properties.

Due to Larva's replication mechanism, the upper-bound memory guarantee is relative to the number of distinct objects being monitored. Consider a scenario where a server, instead of runtime verifying all the executions, delegated the monitoring to the individual clients — each client would be responsible for the correct execution of the code on his/her side. However, if the correct execution of code on the client side affects the behaviour of the server, then we might need some proof that the client actually used runtime verification. There might be several approaches for this problem: (i) provide some mechanism so the client can give evidence that monitoring has truly been carried out; (ii) force the user to supply a log of the client side activities which can be later verified by the server before anything is committed.

By using Lustre to write security-critical properties, we have exploited the language's guarantees on bounded resources. Translating these properties into an existing monitoring framework called Larva makes monitoring of programs both easily applicable to Java programs and at the same time guarantee to use bounded-resources. This has also been extended to handle real-time properties which are particularly sensitive to changes in a system. Furthermore, a subset of QDDC has been suggested as an alternative specification notation for real-time properties because it is translatable into Lustre. Thus, QDDC also enjoys the same guarantees given when using Lustre. A mixture of QDDC and Lustre was subsequently used for a case-study to demonstrate the practicality of using the proposed framework.

The few examples in the case study have the aim to illustrate how non-trivial properties can be represented relatively easily through Lustre or QDDC. Although there are some limitation as regards to the expressivity of both Lustre and QDDC, the framework presented is quite flexible and additional constructs can easily be introduced. This extensibility allows for more complex properties to be expressed using few high-level constructs, which in turn can be defined using other lower-level one. This idea is noteworthy given the fact that the properties for runtime verification should be clearly and succinctly expressible so that the chance of error is minimised.

Undoubtedly, the overhead introduced by runtime verification is a concern, so hopefully resource upper-bound guarantees would help to heighten the confidence of the developers. Another recurrent issue is the identification of appropriate notations to represent properties. We tried to tackle this problem by offering a range of formalisms which are expressive enough to handle most real-time properties.

References

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[AYR95] A. Bouajjani, Y. Lakhnech, and R. Robbana. From duration calculus to linear hybrid automata. In *CAV'95*, volume 939, pages 196–210, Belgium, 1995.

[Col08] Christian Colombo. Practical runtime monitoring with impact guarantees of java programs with real-time constraints. Master's thesis, University of Malta, 2008.

[CPHP87] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. Lustre: a declarative language for programming synchronous systems. In *POPL '87*, pages 178–188, 1987.

[CPS08] Christian Colombo, Gordon J. Pace, and Gerardo Schneider. Dynamic event-based runtime monitoring of real-time and contextual properties. In *Formal Methods for Industrial Critical Systems (FMICS)*, volume 5596 of *Lecture Notes in Computer Science*, pages 135–149, L'Aquila, Italy, 2008.

[DSS+05] Ben D'Angelo, Sriram Sankaranarayanan, César Sánchez, Will Robinson, Bernd Finkbeiner, Henny B. Sipma, Sandeep Mehrotra, and Zohar Manna. Lola: Runtime monitoring of synchronous systems. In *TIME'05*, pages 166–174. IEEE Computer Society Press, June 2005.

[GHR06] Laure Gonnord, Nicolas Halbwachs, and Pascal Raymond. From discrete duration calculus to symbolic automata. *Electr. Notes Theor. Comput. Sci.*, 153(4):3–18, 2006.

[HC97] Michael R. Hansen and Zhou Chaochen. Duration calculus: Logical foundations. *Formal Asp. Comput.*, 9(3):283–330, 1997.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language lustre. *Proceedings of the IEEE*, 79:1305–1320, 1991.

[HLR92] Nicolas Halbwachs, Fabienne Lagnier, and Christophe Ratel. Programming and verifying real-time systems by means of the synchronous dataflow language lustre. *IEEE Trans. Softw. Eng.*, 18(9):785–793, 1992.

[HLR93] Nicolas Halbwachs, Fabienne Lagnier, and Pascal Raymond. Synchronous observers and the verification of reactive systems. In *AMAST*, pages 83–96, 1993.

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP'97*, pages 220–242, 1997.

[Pan01] P.K. Pandya. Specifying and deciding quantified discrete-time duration calculus formulae using dcvalid. In *RTTOOLS'2001*, Aalborg, August 2001.

[ZCA91] Z. ChaoChen, C.A.R. Hoare, and A.P. Ravn. A calculus of durations. *Information Processing Letters*, 40(5):269–276, 1991.

View publication stats