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

Christian Colombo
Gordon J. Pace
Gerardo Schneider

FMICS - September 2008
Why Runtime Verification?

- In security-critical systems we cannot afford to fail!
- Model checking is not scalable.
- Testing lacks coverage.
- Runtime verification is a good compromise.
- There is a great gap between the design and the implementation.
General Architecture

MONITORED SYSTEM

FEEDBACK

EVENTS

VERIFYING SYSTEM

-----------------
Specification
-----------------
Simple Examples

- Ensuring that only authorised users access reserved areas in the system.
- Checking that a train gate which started closing has indeed closed after a number of seconds.
- Monitoring the life-cycle of an object (such as a transaction), ensuring it goes through its stages properly.
Specifying Properties

- Intuitive, clear and succinct logic.
- Understandable and useable by developers.
- Includes all the required expressive power.
- Automatically instrumentable in the target system.
- Low overheads (eg. Determinism)
Dynamic Automata with Timers & Events (DATE)

- Communicating symbolic automata enriched with **events** and **timers**.
- Automata are automatically replicated according to context: hence **dynamic**.
- Supports:
  - Conditions and actions on transitions
  - Real-time
  - Communication between automata
A Scenario - Events

No 3 successive bad logins
A Scenario – Conditions & Actions

No 3 successive bad logins

Logged in

Logged out

Bad login

Good login

logout

Bad login

Logged out

Logged out

Bad login

Logged out

Bad login

Logged out

Bad login

logged out

Bad logins

Bad login / ValidAccount() / BlockAccount();
A Scenario - Clocks

Logged out

logout

Goodlogin

Logged in

Interact \ t.reset();

t@30

Inactive

Logged out

Goodlogin

Logged out

Goodlogin

Logged out

Goodlogin

Badlogins

Badlogin

Badlogin

Badlogin

Badlogin

Logged out

Logged out

Logged out

Logged out

Logged out
A Scenario – Channels

Load Site

Prompt for PW

Press OK \ checkUserName()

Press OK \ checkPassword()

\ ChGoodlogin !

Logged in

Logged out

ChGoodlogin?

Badlogin

Logged out

Logged out

ChGoodlogin?

Logged out

Badlogin

Logged out

Badlogin

Badlogin

Badlogin

Badlogins
A Scenario – Dynamic Triggers

- Imagine we need to check login/logout for each user.
- We have to **trigger** an automaton for every user, to keep track whether each user is logged in or not.
- Use method parameters to get **context**.
Specifying Context

- Actions and conditions on transitions can access the context (User).
- A context can be nested to have a more specific context within it:
  - Eg: Check login for each **site** of each individual **user**.
A Scenario – Context

Trigger new automaton FOR EACH user

Load Site 

PressOK \ checkUserName()

Prompt for PW 

PressOK \ checkPassword() \ Goodlogin !

Good Login

Trigger new automaton FOR EACH user

Logged out

Logged out

Logged out

Logged out

Logged out

Logged out

Logged out

Bad login

Bad login

Bad login

Bad login

Bad login

Bad login

Bad logins

ChGoodlogin?

ChGoodlogin?

ChGoodlogin?

ChGoodlogin?
LARVA - Architecture
LARVA - Architecture (2)

COMPILER

--- EVENTS & PROPERTIES ---

LARVA

---

AspectJ Matching method names

MONITORING SYSTEM

(SYMBOLIC AUTOMATON)
LARVA - Architecture (3)
LARVA - Compilation into Java

- AOP to capture events.
- A hierarchy of classes: one for each context.
- Each class has a reference to its parent context. (E.g. The account context, have access to the user context.)
- A hashmap to keep track of the distinct objects which we are checking.
Recall Scenario

Trigger new automaton FOREACH user

Load Site

Prompt for PW

PressOK \ checkUserName()\

PressOK \ checkPassword() \ Goodlogin !

Good Login

Recall Scenario

Logged out

Logged in

Logged out

Logged out

Logged out

Badlogin

Badlogin

Badlogin

Badlogin

Badlogins

PressOK

PressOK

PressOK

PressOK

FOREACH user

FOREACH user

ChGoodlogin?

ChGoodlogin?

ChGoodlogin?

ChGoodlogin?
LARVA – Script

GLOBAL { FOREACH (User u) {
    VARIABLES { Channel gl; }
    EVENTS {
        goodlogin() = {gl.receive(User u1)} where {u = u1;}
        pressOK() = {*.pressedOK(u1)} where {u = u1;}
        badlogin() = {*.loginTry(u1)} where {u = u1;}
    }
    PROPERTY one {
        STATES {
            BAD { badlogins }
            NORMAL { loggedout2 loggedout3 loggedin }
            STARTING { loggedout1 }
        }
        TRANSITIONS {
            loggedout1 -> loggedin [goodlogin]
            loggedout2 -> loggedin [goodlogin]
            loggedout3 -> loggedin [goodlogin]
            loggedout1 -> loggedout2 [badlogin]
            loggedout2 -> loggedout3 [badlogin]
            loggedout3 -> badlogins [badlogin]
        }
    }
    PROPERTY two {
        STATES {
            NORMAL { promptPW goodlogin }
            STARTING { loadsite }
        }
        TRANSITIONS {
            loadsite -> promptPW [PressOK\checkUserName()]
            promptPW -> goodlogin [PressOK\checkPassword()\gl.send(u);]
            promptPW -> loadsite [PressOK]
        }
    }
} }

METHODS {
    boolean checkUserName(){return true;}
    boolean checkPassword(){return true;}
}
Case-Study (1): Credit Card System

- Relatively large system (>26 kloc)
- Great security implications
- Challenges:
  - Communication with 3rd party systems
  - Although deployed, the system had no proper documented specification
Case-Study (2): Properties

- Logging of credit card numbers – no risk of exposing sensitive information.
- Execution of transactions – correct progress through states.
- Authorisation transaction – transaction consistency.
- Backlog – retries in case of failure.
Case-Study (3): - Experience

- A lot of interesting properties are relatively simple.
- Intuitive definition of properties.
- Identified shortcomings of Larva and it was extended.
- RV helps in clearly identifying requirements.
- Integration in system life cycle.
# Benchmark – Expressivity

Table 1. Expressivity features of various tools.

<table>
<thead>
<tr>
<th>Tool</th>
<th>LARVA</th>
<th>ConSpec</th>
<th>Java-MOP</th>
<th>Java-MaC</th>
<th>Hawk</th>
<th>Lola</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exceptions</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Temporal Logics</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Real-Time</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>✓&lt;sup&gt;b&lt;/sup&gt;</td>
<td>✓&lt;sup&gt;c&lt;/sup&gt;</td>
<td>×</td>
</tr>
<tr>
<td>Mobile Application Policies</td>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Invariants</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Feedback</td>
<td>✓</td>
<td>Stop. ✓</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Conditions</td>
<td>✓</td>
<td>✓</td>
<td>✓&lt;sup&gt;d&lt;/sup&gt;</td>
<td>✓</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Numerical Queries</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>✓</td>
</tr>
</tbody>
</table>

<sup>a</sup> in specification it supports all the mentioned scopes but currently only session is supported

<sup>b</sup> restricted (cannot trigger clock events)

<sup>c</sup> can be extended to support real-time

<sup>d</sup> restricted to implementing conditions in violation/validation handling method
Benchmark – Performance

- Dummy transaction processing system (4 properties – 2 real-time)
- Memory and time required is considerable but linear to the number of objects being monitored (replication of automata).
- Compares well with Java-MOP which is the most similar work available for usage.
Ongoing Work

- Translation from other logics to DATEs
- Guaranteeing time and memory upperbounds
  - Going through Lustre
  - Starting from a subset of QDDC
- Guaranteeing effect of runtime verification on real-time properties upon adding/removing monitors
Conclusions

- Mathematical framework – DATE
- Implemented usable tool – LARVA
- Highly expressive (incl. real-time)
- Used in an industrial case-study
- Evolving theory with practical guarantees
Questions

?