Towards Efficient and Expressive Runtime Monitors
Understanding Complex Systems by Monitoring their Execution

Klaus Havelund
Jet Propulsion Laboratory
California Institute of Technology

April 20, 2012
In Collaboration With

- University of Manchester, UK
  - Howard Barringer (Professor)
  - Giles Reger (Ph.D. student)
  - David Rydeheard (Dr.)

- University of Grenoble, France
  - Yliès Falcone (Associate Professor)
Contents

• What Runtime Verification is
• From propositional to parametric properties
• Quantified Event Automata (an automaton approach)
• TraceContract (a formula rewriting approach)
• ScalaRules (a production rule system approach)
• Conclusion
Runtime Verification

- Monitoring the runtime behavior of a system with respect to a user-defined property
- Need to instrument system to select relevant events
- Online or offline (log-files)
- If online
  - verdict returned after each event
  - can give feedback to steer the system
Runtime Verification in Theory

- *Events* record runtime behavior
  - snapshots of state or actions performed
- A finite sequence of events is a *trace* $\tau$
- A *property* $\phi$ denotes an *event language* $\mathcal{L}(\phi)$ (a set of traces)
- $\tau$ satisfies $\phi$ iff $\tau \in \mathcal{L}(\phi)$
Runtime Verification in Theory

- Should detect success/failure before end of trace
- Standard approach is to use *four-valued verdict domain*
- Consider all possible extensions of a trace

<table>
<thead>
<tr>
<th>current trace $\tau$</th>
<th>all suffixes $\sigma$</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1$</td>
<td>$\tau \in \mathcal{L}(\varphi)$</td>
<td>$\tau\sigma \in \mathcal{L}(\varphi)$</td>
</tr>
<tr>
<td>$2$</td>
<td>$\tau \in \mathcal{L}(\varphi)$</td>
<td>unknown</td>
</tr>
<tr>
<td>$3$</td>
<td>$\tau \notin \mathcal{L}(\varphi)$</td>
<td>$\tau\sigma \notin \mathcal{L}(\varphi)$</td>
</tr>
<tr>
<td>$4$</td>
<td>$\tau \notin \mathcal{L}(\varphi)$</td>
<td>unknown</td>
</tr>
</tbody>
</table>
Runtime Verification in Practice

- Start with a system to monitor
Runtime Verification in Practice

- *Instrument* the system to record relevant events
Runtime Verification in Practice

• *Generate* a monitor from the property
Runtime Verification in Practice

- *Dispatch* each received event to the monitor.
Runtime Verification in Practice

- Compute a *verdict* for the trace received so far.
Runtime Verification in Practice

- Possibly generate *feedback* to the system.
Runtime Verification Applications

- Detect erroneous behavior after deployment (fault protection)
- Detect intrusion after deployment (security)
- Monitor as part of testing before deployment (test oracles)
- Program understanding
Some Context

- Field started with *propositional* monitoring
  - events are just strings
- Recently moved to *parametric* monitoring
  - events include data values
- Solutions exist spanning the two classical dimensions
  - *Expressiveness* of specification language
  - *Efficiency* of monitoring algorithm
- This work is looking for the right combination
The Propositional Approach : An Example

- Record *propositional* events, for example
  - open, close
- Define a property over propositional events, for example
  - LTL (finite-trace) \( \Box(\text{open} \rightarrow \Diamond(\neg \text{open} \lor \text{close})) \)
  - RE
    \[
    (\text{open.close})^*
    \]
  - DFA
- Check if each trace prefix is in the language of the property
• Consider the code

File f1 = new File("manual.pdf");
File f2 = new File("readme.txt");
f1.open();
f2.open();
f2.close();
f1.close();
Going Parametric

- Consider the code
  ```java
  File f1 = new File("manual.pdf");
  File f2 = new File("readme.txt");
  f1.open();
  f2.open();
  f2.close();
  f1.close();
  ```

- Say we just focus on propositional events
  ```java
  open.open.close.close
  ```
Going Parametric

- Consider the code
  
  ```java
  File f1 = new File("manual.pdf");
  File f2 = new File("readme.txt");
  f1.open();
  f2.open();
  f2.close();
  f1.close();
  ```

- Say we just focus on propositional events

  `open.open.close.close`

- No good, we want to *parameterize* events with data values and use those values in the specification
Going Parametric

- Consider the code
  ```
  File f1 = new File("manual.pdf");
  File f2 = new File("readme.txt");
  f1.open();
  f2.open();
  f2.close();
  f1.close();
  ```
- Say we just focus on propositional events
  `open.open.close.close`
- No good, we want to *parameterize* events with data values and use those values in the specification
- Instead record the parametric trace
Parametric Properties

- Using the events
  - `open(f)` when file f is opened
  - `close(f)` when file f is closed
Parametric Properties

- Using the events
  - `open(f)` when file f is opened
  - `close(f)` when file f is closed
- the property becomes

![Diagram showing states 1 and 2 with transitions `open(f)` and `close(f)`]
Instantiating Parametric Property

- Let $f = \text{readme.txt} \ (a \ binding)$
Instantiating Parametric Property

- Let \( f = \text{readme.txt} \) (a binding)
- Instantiated property becomes

\[
\begin{align*}
\text{open(readme.txt)} \\
\text{close(readme.txt)}
\end{align*}
\]
Instantiating Parametric Property

- Let $f = \text{readme.txt}$ (a binding)
- Instantiated property becomes

  $\text{open}(\text{readme.txt})$

  1

  $\text{close}(\text{readme.txt})$

  2

- Given parametric trace

  $\text{open}(\text{manual.pdf}).\text{open(\text{readme.txt})}.\text{close(\text{readme.txt})}.\text{close(\text{manual.pdf})}$
Instantiating Parametric Property

- Let $f = \text{readme.txt}$ (a binding)
- Instantiated property becomes

$$\text{open(\text{readme.txt})}$$

$$\text{close(\text{readme.txt})}$$

- Given parametric trace

$$\text{open(\text{manual.pdf}).open(\text{readme.txt}).close(\text{readme.txt}).close(\text{manual.pdf})}$$

- project to

$$\text{open(\text{readme.txt}).close(\text{readme.txt})}$$
From Parametric to Quantified

• Where do bindings come from?
From Parametric to Quantified

- Where do bindings come from?
- quantify over variables in parametric property
From Parametric to Quantified

- Where do bindings come from?
- quantify over variables in parametric property

∀f

open(f)  close(f)

1  2
From Parametric to Quantified

- Where do bindings come from?
- quantify over variables in parametric property

\[ \forall f \]

\[ \text{open}(f) \]

1 \rightarrow 2

\[ \text{close}(f) \]

- Universal and existential quantification
From Parametric to Quantified

- Where do bindings come from?
- quantify over variables in parametric property

\[
\forall f \quad \text{open}(f) \quad \text{close}(f)
\]

- Universal and existential quantification
- What is the domain of quantification? (choice)
From Parametric to Quantified

- Where do bindings come from?
- quantify over variables in parametric property

\[ \forall f \rightarrow \text{open}(f) \rightarrow \text{close}(f) \]

- Universal and existential quantification
- What is the domain of quantification? (choice)
- Extract domain of quantification from trace
From Parametric to Quantified

- Where do bindings come from?
- quantify over variables in parametric property

\[ \forall f \text{ open}(f) \text{ close}(f) \]

- Universal and existential quantification
- What is the domain of quantification? (choice)
- Extract domain of quantification from trace
- How? (choice)
From Parametric to Quantified

- Where do bindings come from?
- quantify over variables in parametric property

\[ \forall f \]

open(f) \hspace{2cm} close(f)

- Universal and existential quantification
- What is the domain of quantification? (choice)
- Extract domain of quantification from trace
- How? (choice)
- Match events in parametric property with events in trace
From Parametric to Quantified

- Where do bindings come from?
- quantify over variables in parametric property

\[ \forall f \text{ open}(f) \rightarrow \text{close}(f) \]

- Universal and existential quantification
- What is the domain of quantification? (choice)
- Extract domain of quantification from trace
- How? (choice)
- Match events in parametric property with events in trace
- \text{open}(f) matches \text{open}(\text{readme.txt}) and \text{open}(\text{manual.pdf})

\[ [f \mapsto \{ \text{readme.txt, manual.pdf} \}] \]
Using Data Values: A Task Monitoring Example

• All tasks must end phases in increasing order
Using Data Values : A Task Monitoring Example

- All tasks must end phases in increasing order
- Events monitored: end(task, phase)
Using Data Values: A Task Monitoring Example

- All tasks must end phases in increasing order
- Events monitored: \(\text{end}(\text{task}, \text{phase})\)
- Example trace
  \[\text{end}(42, 5).\text{end}(42, 6).\text{end}(42, 3)\]
Using Data Values: A Task Monitoring Example

- All tasks must end phases in increasing order
- Events monitored: end(task, phase)
- Example trace
  end(42, 5).end(42, 6).end(42, 3)
Using Data Values: A Task Monitoring Example

\[ \forall item \]

\[ \text{end}(task, max) \quad \text{end}(task, new) \quad \text{new} \leq \text{max} \]

\[ \text{new} > \text{max} \quad \text{max} := \text{new} \]

- trace:  \text{end}(42, 5).\text{bid}(42, 6).\text{bid}(42, 3)
Using Data Values: A Task Monitoring Example

\[ \forall item \]

\[ \text{end}(task, max) \]

\[ \text{end}(task, new) \]

\[ \text{new} \leq \text{max} \]

\[ \text{new} > \text{max} \]

\[ \text{max} := \text{new} \]

- trace: \text{end}(42, 5).\text{bid}(42, 6).\text{bid}(42, 3)
- domain is \( task \mapsto \{ 42 \} \)
Using Data Values: A Task Monitoring Example

∀item

\[
\begin{align*}
\text{end}(\text{task}, \text{max}) & \quad \text{end}(\text{task}, \text{new}) & \quad \text{end}(\text{task}, \text{new}) \\
1 & \rightarrow 2 & \quad \frac{\text{new} \leq \text{max}}{\text{new} > \text{max}} \\
\end{align*}
\]

- trace: end(42, 5).bid(42, 6).bid(42, 3)
- domain is [task \mapsto \{42\}]
- partially instantiate parametric property with [task \mapsto 42]
Using Data Values: A Task Monitoring Example

\[ \forall \text{item} \]

\[ \text{end}(\text{task}, \text{max}) \rightarrow 3 \]

\[ \text{end}(\text{task}, \text{new}) \quad \text{new} \leq \text{max} \]

\[ \text{new} > \text{max} \quad \text{max} := \text{new} \]

- trace: \( \text{end}(42, 5).\text{bid}(42, 6).\text{bid}(42, 3) \)
- domain is \([\text{task} \mapsto \{42\}]\)
- partially instantiate parametric property with \([\text{task} \mapsto 42]\)
- to get Event Automaton

\[ \text{end}(42, \text{max}) \rightarrow 1 \]

\[ \text{end}(42, \text{new}) \quad \text{new} \leq \text{max} \]

\[ \text{new} > \text{max} \quad \text{max} := \text{new} \]
Using Data Values: A Task Monitoring Example

- trace: end(42, 5).bid(42, 6).bid(42, 3)
- domain is \([task \mapsto \{42\}]\)
- partially instantiate parametric property with \([task \mapsto 42]\)
- to get Event Automaton

- keep local state per instantiated parametric property
Using Data Values: A Task Monitoring Example

\[ \forall \text{item} \]

- trace: \( \text{end}(42, 5).\text{bid}(42, 6).\text{bid}(42, 3) \)
- domain is \([\text{task} \mapsto \{ 42 \}]\)
- partially instantiate parametric property with \([\text{task} \mapsto 42]\)
- to get Event Automaton

- keep local state per instantiated parametric property
- treat quantified and unquantified variables differently
Another Example

- Every object can only be locked once at any one time

\[ \forall \text{thread}, \forall \text{obj} \]

\[ \text{lock} (\text{thread}, \text{obj}) \]

\[ \text{unlock} (\text{thread}, \text{obj}) \]

\[ \text{lock} (\text{thread}_2, \text{obj}) \]
Another Example

• Every object can only be locked once at any one time

∀thread, ∀obj

lock(thread, obj)

unlock(thread, obj)

lock(thread₂, obj)

• lock is used with two different lists of formal parameters
Event Automata: Definition

$Bind \ = \ [Var \rightarrow Val]$

$Guard \ = \ [Bind \rightarrow \mathbb{B}]$

$Assign \ = \ [Bind \rightarrow Bind]$

Definition (Event Automaton)

An Event Automaton $\langle Q, A, \delta, q_0, F \rangle$ is a tuple where

- $Q$ is the set of states,
- $A \subseteq \text{Event}$ is the alphabet,
- $\delta \in (Q \times A \times Guard \times Assign \times Q)$ is the transition set,
- $q_0$ is the initial state, and
- $F \subseteq Q$ is the set of final states.
Quantified Event Automata: Definition

Definition (Quantified Event Automaton)

A QEA is a pair \( \langle \Lambda, E \rangle \) where

- \( E \) is an EA, and
- \( \Lambda \in (\{\forall, \exists\} \times \text{variables}(E) \times \text{Guard})^* \) is a list of quantified variables with guards.
• Previous description is big-step (whole trace)
Monitoring QEA

- Previous description is big-step (whole trace)
- Want to process a trace an event at a time
Monitoring QEA

- Previous description is big-step (whole trace)
- Want to process a trace an event at a time
- Small-step monitoring construction
Monitoring QEA

- Previous description is big-step (whole trace)
- Want to process a trace an event at a time
- Small-step monitoring construction
  - Build up domain on the fly
  - Generate new bindings on the fly
  - Track configurations associated with bindings
  - Check acceptance
Monitoring QEA

- Previous description is big-step (whole trace)
- Want to process a trace an event at a time
- Small-step monitoring construction
  - Build up domain on the fly
  - Generate new bindings on the fly
  - Track configurations associated with bindings
  - Check acceptance
- Efficient algorithms
Monitoring QEA

- Previous description is big-step (whole trace)
- Want to process a trace an event at a time
- Small-step monitoring construction
  - Build up domain on the fly
  - Generate new bindings on the fly
  - Track configurations associated with bindings
  - Check acceptance
- Efficient algorithms
  - Lookup relevant monitors from event
  - Data-structures to deal with *matching*
TraceContract

- An internal Scala DSL (API)
TraceContract

• An internal Scala DSL (API)
• Expressive and easy to implement and modify
TraceContract

- An internal Scala DSL (API)
- Expressive and easy to implement and modify
- Based on formula rewriting: \( p \cup q = q \lor (p \land \Box (p \cup q)) \)
TraceContract

- An internal Scala DSL (API)
- Expressive and easy to implement and modify
- Based on formula rewriting: $p \cup q = q \lor (p \land \circ (p \cup q))$
- (To be) used by LADEE: Lunar Atmosphere and Dust Environment Explorer
trait Event

case class End(task: Int, step: Int) extends Event

class TaskMonitor extends Monitor[Event] {
  always {
    case End(task, step1) =>
      watch {
        case End('task', step2) => step2 > step1
      }
  }
}

Analyzing a Trace

```scala
object Test extends Application {
  val m = new TaskMonitor

  val trace = List(
    End(1, 2),
    End(2, 1),
    End(1, 3),
    End(2, 2),
    End(1, 1))

  m.verify(trace)
}
```
Implementation of TraceContract

```scala
trait Monitor[E] extends RuleSystem {
  var current: state = True

  trait state {
    def apply(e: E): state
    def and(that: state) = And(this, that) reduce
    def or(that: state) = Or(this, that) reduce
  }

  type Body = PartialFunction[E, state]

  case class watch(b: Body) extends state {
    def apply(e: E) = if (b.isDefinedAt(e)) b(e) else this
  }

  ...
}
```
Implementation using Rewriting

case class repeat(b: Body) extends state {
  def apply(e: E) =
    if (b.isDefinedAt(e)) And(b(e), this) reduce else this
}

def init(s: state) {current = s}

def always(b: Body) = init(repeat(b))

def apply(e: E) {
  current = current(e)
  if (current == False) println("*** safety violation")
}
ScalaRules

- An internal Scala DSL (API)
ScalaRules

- An internal Scala DSL (API)
- Implements the RETE algorithm for rule-based systems
ScalaRules

- An internal Scala DSL (API)
- Implements the RETE algorithm for rule-based systems
- Efficient pattern matching algorithm for production rule systems
ScalaRules

- An internal Scala DSL (API)
- Implements the RETE algorithm for rule-based systems
- Efficient pattern matching algorithm for production rule systems
- Purpose is to investigate relevance for runtime verification
class LockMonitor extends ScalaRules {

rule ("goodLock") when
  exists ('kind->"lock", 'thread->'t, 'obj->'o) then
  add('kind->"Locked", 'thread->'t, 'obj->'o)

rule ("badLock") when
  exists ('kind->"lock", 'thread->'t1, 'obj->'o) and
  exists ('kind->"Locked", 'thread->'t2, 'obj->'o) then
  error

rule ("unlock") when
  exists ('kind->"unlock", 'thread->'t, 'obj->'o) and
  exists ('x)('kind->"Locked", 'thread->'t, 'obj->'o) then
  rem('x)
}
object Test extends Application {
  val r = new LockMonitor
  r.addFact('kind -> "lock", 'thread -> 1, 'obj -> 42)
  r.addFact('kind -> "unlock", 'thread -> 1, 'obj -> 42)
  r.addFact('kind -> "lock", 'thread -> 1, 'obj -> 42)
  r.addFact('kind -> "lock", 'thread -> 2, 'obj -> 42)
}
RETE Network for a Rule

rule: $a(x), b(x, y), c(x, y) \Rightarrow action$
Future Work

• Theoretic foundations
  • QEA
  • TraceContract

• Implementation
  • Implement efficient monitoring algorithm for QEA
  • Explore utility of RETE algorithm and modifications

• Application
  • Support application of TraceContract within LADEE mission
  • Apply to logs for JPL mission

• Related topics
  • Inferring properties from runs
  • Annotating logs (Rajeev Joshi)
  • Program visualization
Conclusion

- Efficient state of the art systems lack expressiveness
- We attempt to increase expressiveness while staying efficient
- Result should be efficient and expressive RV system
- RV is a scalable way to understand complex systems
- Scala as prototyping language + internal DSLs speed up development
Publications

- **Aspect-Oriented Instrumentation with GCC**  
  RV 2010, St. Julians, Malta.

- **TraceContract: A Scala DSL for Trace Analysis**  
  H. Barringer and K. Havelund  
  FM 2011, Limerick, Ireland

- **Runtime Verification with State Estimation**  
  S. D. Stoller, E. Bartocci, J. Seyster, R. Grosu, K. Havelund, S. A. Smolka, and E. Zadok  
  RV 2011, San Francisco, California, USA.

- **Quantified Event Automata: Towards Expressive and Efficient Runtime Monitors**  
  H. Barringer, Y. Falcone, K. Havelund, G. Reger, and D. Rydeheard  
  Submitted for publication, under review. March 2012.