Runtime Verification

Martin Leucker

Institute for Software Engineering
Universität zu Lübeck

VTSA 2023 - Runtime Verification
Runtime Verification (RV)
Runtime Verification (RV)

\[ M \]

\[ S_1 \rightarrow S_3 \rightarrow S_4 \rightarrow S_2 \]
always (not $x > 0$ implies next $x > 0$)
always (not $x > 0$ implies next $x > 0$)

Characterisation

- Verifies (partially) correctness properties based on actual executions
**Runtime Verification (RV)**

always \((\text{not } x > 0 \text{ implies next } x > 0)\)

\[ M \]

**Characterisation**

- Verifies (partially) correctness properties based on actual executions
- **Simple** verification technique
always (not $x > 0$ implies next $x > 0$)

Characterisation

- Verifies (partially) correctness properties based on actual executions
- Simple verification technique
- Complementing
always (not $x > 0$ implies next $x > 0$)

Characterisation

- Verifies (partially) correctness properties based on actual executions
- Simple verification technique
- Complementing
  - Model Checking
Runtime Verification (RV)

always (not $x > 0$ implies next $x > 0$)

Characterisation

- Verifies (partially) correctness properties based on actual executions
- **Simple** verification technique
- Complementing
  - Model Checking
  - Testing
Runtime Verification (RV)

always (not $x > 0$ implies next $x > 0$)

Characterisation

- Verifies (partially) correctness properties based on actual executions
- Simple verification technique
- Complementing
  - Model Checking
  - Testing
- Formal: $w \in \mathcal{L}(\varphi)$
Specification of System
Model Checking

- Specification of System
  - as formula $\varphi$ of linear-time temporal logic (LTL)
Model Checking

- Specification of System
  - as formula $\varphi$ of linear-time temporal logic (LTL)
  - with models $\mathcal{L}(\varphi)$
Model Checking

- Specification of System
  - as formula $\varphi$ of linear-time temporal logic (LTL)
  - with models $\mathcal{L}(\varphi)$

- Model of System
Model Checking

- Specification of System
  - as formula $\varphi$ of linear-time temporal logic (LTL)
  - with models $\mathcal{L}(\varphi)$

- Model of System
  - as transition system $S$ with runs $\mathcal{L}(S)$
Model Checking

- Specification of System
  - as formula $\varphi$ of linear-time temporal logic (LTL)
  - with models $L(\varphi)$

- Model of System
  - as transition system $S$ with runs $L(S)$

- Model Checking Problem:
  Do all runs of the system satisfy the specification
Model Checking

- Specification of System
  - as formula $\varphi$ of linear-time temporal logic (LTL)
  - with models $\mathcal{L}(\varphi)$

- Model of System
  - as transition system $S$ with runs $\mathcal{L}(S)$

- Model Checking Problem:
  Do all runs of the system satisfy the specification
  - $\mathcal{L}(S) \subseteq \mathcal{L}(\varphi)$
Model Checking versus RV

- Model Checking: infinite words
Model Checking versus RV

- Model Checking: infinite words
- Runtime Verification: finite words
Model Checking versus RV

- Model Checking: infinite words
- Runtime Verification: finite words
  - yet continuously expanding words
Model Checking versus RV

- Model Checking: infinite words
- Runtime Verification: finite words
  - yet continuously expanding words
- In RV: Complexity of monitor generation is of less importance than complexity of the monitor
Model Checking versus RV

- Model Checking: infinite words
- Runtime Verification: finite words
  - yet continuously expanding words
- In RV: Complexity of monitor generation is of less importance than complexity of the monitor
- Model Checking: White-Box-Systems
Model Checking versus RV

- Model Checking: **infinite words**
- Runtime Verification: **finite words**
  - yet **continuously expanding words**
- In RV: Complexity of monitor generation is of less importance than complexity of the monitor
- Model Checking: **White-Box-Systems**
- Runtime Verification: also **Black-Box-Systems**
Testing: Input/Output Sequence

- **incomplete** verification technique
Testing: Input/Output Sequence

- **Incomplete** verification technique
- **Test case**: finite sequence of input/output actions
Testing: Input/Output Sequence

- **incomplete** verification technique
- **test case**: finite sequence of input/output actions
- **test suite**: finite set of test cases
Testing: Input/Output Sequence

- **incomplete** verification technique
- **test case**: finite sequence of input/output actions
- **test suite**: finite set of test cases
- **test execution**: send inputs to the system and check whether the actual output is as expected
Testing: Input/Output Sequence

- **incomplete** verification technique
- **test case**: finite sequence of input/output actions
- **test suite**: finite set of test cases
- **test execution**: send inputs to the system and check whether the actual output is as expected
Testing: Input/Output Sequence

- **incomplete** verification technique
- **test case**: finite sequence of input/output actions
- **test suite**: finite set of test cases
- **test execution**: send inputs to the system and check whether the actual output is as expected

Testing: with Oracle

- **test case**: finite sequence of input actions
Testing: Input/Output Sequence

- **incomplete** verification technique
- **test case**: finite sequence of input/output actions
- **test suite**: finite set of test cases
- **test execution**: send inputs to the system and check whether the actual output is as expected

Testing: with Oracle

- **test case**: finite sequence of input actions
- **test oracle**: monitor
Testing

Testing: Input/Output Sequence

- **incomplete** verification technique
- **test case**: finite sequence of input/output actions
- **test suite**: finite set of test cases
- **test execution**: send inputs to the system and check whether the actual output is as expected

Testing: with Oracle

- **test case**: finite sequence of input actions
- **test oracle**: monitor
- **test execution**: send test cases, let oracle report violations
Testing: Input/Output Sequence

- **incomplete** verification technique
- **test case**: finite sequence of input/output actions
- **test suite**: finite set of test cases
- **test execution**: send inputs to the system and check whether the actual output is as expected

Testing: with Oracle

- **test case**: finite sequence of input actions
- **test oracle**: monitor
- **test execution**: send test cases, let oracle report violations
- **similar to runtime verification**
Testing versus RV

- Test oracle manual
Testing versus RV

- Test oracle manual
- RV monitor from high-level specification (LTL)
Testing versus RV

- Test oracle manual
- RV monitor from high-level specification (LTL)
- Testing:
  How to find good test suites?
Testing versus RV

- Test oracle **manual**
- RV monitor **from high-level specification (LTL)**
- Testing: 
  *How to find good test suites?*
- Runtime Verification:
  *How to generate good monitors?*
Outline

Runtime Verification

Runtime Verification for LTL

- LTL over Finite, Completed Words
- LTL over Finite, Non-Completed Words: Impartiality
- LTL over Non-Completed Words: Anticipation
- Monitorable Properties
- RV-LTL
- LTL with a Predictive Semantics
- LTL wrap-up

Extensions

Monitor Systems/Logging

Steering

RV frameworks

JUnit\textsuperscript{RV} – Testing Temporal Properties

- Motivating Example
- jUnit\textsuperscript{RV} – Idea
- Using jUnit\textsuperscript{RV}
Presentation outline

Runtime Verification

Runtime Verification for LTL
- LTL over Finite, Completed Words
- LTL over Finite, Non-Completed Words: Impartiality
- LTL over Non-Completed Words: Anticipation
- Monitorable Properties
- RV-LTL
- LTL with a Predictive Semantics
- LTL wrap-up

Extensions
- Monitoring Systems/Logging
- Steering
- RV frameworks
- jUnit$^\text{RV}$ – Testing Temporal Properties
  - Motivating Example
  - jUnit$^\text{RV}$ – Idea
  - Using jUnit$^\text{RV}$
Runtime Verification

**Definition (Runtime Verification)**

Runtime verification is the discipline of computer science that deals with the study, development, and application of those verification techniques that allow checking whether a run of a system under scrutiny (SUS) satisfies or violates a given correctness property.

Its distinguishing research effort lies in synthesizing monitors from high level specifications.
Runtime Verification

**Definition (Runtime Verification)**

Runtime verification is the discipline of computer science that deals with the study, development, and application of those verification techniques that allow checking whether a run of a system under scrutiny (SUS) satisfies or violates a given correctness property.

Its distinguishing research effort lies in synthesizing monitors from high level specifications.

**Definition (Monitor)**

A monitor is a device that reads a finite trace and yields a certain verdict.

A verdict is typically a truth value from some truth domain.
Presentation outline

Runtime Verification

Runtime Verification for LTL
  LTL over Finite, Completed Words
  LTL over Finite, Non-Completed Words: Impartiality
  LTL over Non-Completed Words: Anticipation
  Monitorable Properties
  RV-LTL
  LTL with a Predictive Semantics
  LTL wrap-up

Extensions

Monitoring Systems/Logging

Steering

RV frameworks

jUnit<sup>RV</sup> – Testing Temporal Properties
  Motivating Example
  jUnit<sup>RV</sup> – Idea
  Using jUnit<sup>RV</sup>
Runtime Verification for LTL

Observing executions/runs
Runtime Verification for LTL

Observing executions/runs

Idea

Specify correctness properties in LTL
Runtime Verification for LTL

Observing executions/runs

Idea
Specify correctness properties in LTL

Commercial
Specify correctness properties in Regular LTL
Definition (Syntax of LTL formulae)

Let $p$ be an atomic proposition from a finite set of atomic propositions AP. The set of LTL formulae, denoted with LTL, is inductively defined by the following grammar:

$$\varphi ::= \text{true} \mid p \mid \varphi \lor \varphi \mid \varphi \ U \varphi \mid X\varphi \mid \text{false} \mid \neg p \mid \varphi \land \varphi \mid \varphi \ R \varphi \mid \bar{X}\varphi \mid \neg \varphi$$
Linear-time Temporal Logic (LTL)

Semantics

over $w \in (2^{AP})^\omega = \Sigma^\omega$

\[
\{p, q\} \quad p \quad p \quad q \quad q \quad \ldots
\]
Semantics

over $w \in (2^{AP})^\omega = \Sigma^\omega$

$\{p, q\}$   $p$   $p$   $q$   $q$   $\ldots$   $\models$
Linear-time Temporal Logic (LTL)

Semantics

over $w \in (2^{AP})^\omega = \Sigma^\omega$

$p \quad \neg p \quad p \cup q \quad X(p \cup q)$
Linear-time Temporal Logic (LTL)

Semantics

over \( w \in (2^{AP})^\omega = \Sigma^\omega \)

\[
\begin{array}{cccccc}
\{p, q\} & p & p & q & q & \ldots \\
\end{array}
\]

\( \models \ p_U q \)

\( \models \ X(p_U q) \)
Linear-time Temporal Logic (LTL)

Semantics

over $w \in (2^{AP})^\omega = \Sigma^\omega$

\[
\begin{array}{cccccc}
\{p, q\} & p & p & q & q & \ldots
\end{array}

\models

\begin{array}{ll}
p & \checkmark \\
\neg p & \times \\
pUq & \\
X(pUq) & \\
\end{array}
Linear-time Temporal Logic (LTL)

**Semantics**

over $w \in (2^{	ext{AP}})\omega = \Sigma^\omega$

$\{p, q\} \quad p \quad p \quad q \quad q \quad \ldots \quad \models \quad X(pUq)$

$\models$:
- $p$: ✓
- $¬p$: ✗
- $pUq$: ✓
Linear-time Temporal Logic (LTL)

Semantics

over $w \in (2^{AP})^\omega = \Sigma^\omega$

$$\{p,q\} \quad p \quad p \quad q \quad q \quad \ldots \quad X(pUq) \quad \checkmark$$

$p \quad \checkmark$

$\neg p \quad \times$

$pUq \quad \checkmark$

$X(pUq) \quad \checkmark$
Linear-time Temporal Logic (LTL)

**Semantics**

over \(w \in (2^{AP})^\omega = \Sigma^\omega\)

\[
\begin{align*}
\{p, q\} & \quad p \\
p & \quad p \\
p & \quad q \\
q & \quad q \\
\ldots & \quad X(pUq) \\
\end{align*}
\]

\(\models\)

- \(p\) \quad \checkmark
- \(\neg p\) \quad \times
- \(pUq\) \quad \checkmark
- \(X(pUq)\) \quad \checkmark

**Abbreviation**

\[F\varphi \equiv \text{true}U\varphi\quad G\varphi \equiv \neg F\neg \varphi\]
Linear-time Temporal Logic (LTL)

Semantics

over $w \in (2^{AP})^\omega = \Sigma^\omega$

$\models$

$p \quad \checkmark$
$\neg p \quad \times$
$pUq \quad \checkmark$
$X(pUq) \quad \checkmark$

Abbreviation

$F\varphi \equiv trueU\varphi \quad G\varphi \equiv \neg F\neg\varphi$

Example

$G\neg(\text{critic}_1 \land \text{critic}_2), G(\neg\text{alive} \rightarrow X\text{alive})$
Definition (LTL semantics (traditional))

Semantics of LTL formulae over an infinite word \( w = a_0 a_1 \ldots \in \Sigma^\omega \), where 

\[ w^i = a_ia_{i+1} \ldots \]

- \( w \models true \)
- \( w \models p \) if \( p \in a_0 \)
- \( w \models \neg p \) if \( p \notin a_0 \)
- \( w \models \neg \varphi \) if not \( w \models \varphi \)
- \( w \models \varphi \lor \psi \) if \( w \models \varphi \) or \( w \models \psi \)
- \( w \models \varphi \land \psi \) if \( w \models \varphi \) and \( w \models \psi \)
- \( w \models X\varphi \) if \( w^1 \models \varphi \)
- \( w \models X^\prime \varphi \) if \( w^1 \models \varphi \)
- \( w \models \varphi \cup \psi \) if there is \( k \) with \( 0 \leq k < |w|: w^k \models \psi \) and for all \( l \) with \( 0 \leq l < k \) \( w^l \models \varphi \)
- \( w \models \varphi \cup \psi \) if for all \( k \) with \( 0 \leq k < |w|: (w^k \models \psi \) or there is \( l \) with \( 0 \leq l < k \) \( w^l \models \varphi \)
LTL for the working engineer??

Simple??

“LTL is for theoreticians—but for practitioners?”
LTL for the working engineer??

Simple??
“LTL is for theoreticians—but for practitioners?”

SALT
Structured Assertion Language for Temporal Logic
“Syntactic Sugar for LTL” [Bauer, L., Streit@ICFEM’06]
SALT - http://www.isp.uni-luebeck.de/salt

SALT - Smart Assertion Language for Temporal Logic

Goal

Do you want to specify the behavior of your program in a rigorously yet comfortable manner?
Do you see the benefits of temporal specifications but are bothered by the awkward formalisms available?
Do you want to use

- the power of a Model Checker to improve the quality of your systems or
- the powerful runtime reflection approach for bug hunting and elimination.
Runtime Verification for LTL

Idea

Specify correctness properties in LTL

Definition (Syntax of LTL formulae)

Let $p$ be an atomic proposition from a finite set of atomic propositions $AP$. The set of LTL formulae, denoted with $LTL$, is inductively defined by the following grammar:

$$\varphi ::= true \mid p \mid \varphi \lor \varphi \mid \varphi U \varphi \mid X\varphi \mid false \mid \neg p \mid \varphi \land \varphi \mid \varphi R \varphi \mid \bar{X}\varphi \mid \neg \varphi$$
A lattice is a partially ordered set \((L, \sqsubseteq)\) where for each \(x, y \in L\), there exists

1. a unique greatest lower bound (glb), which is called the meet of \(x\) and \(y\), and is denoted with \(x \sqcap y\), and
2. a unique least upper bound (lub), which is called the join of \(x\) and \(y\), and is denoted with \(x \sqcup y\).

A lattice is called finite iff \(L\) is finite.

Every finite lattice has a well-defined unique least element, called bottom, denoted with \(\bot\), and analogously a greatest element, called top, denoted with \(\top\).
A lattice is distributive, iff $x \sqcap (y \sqcup z) = (x \sqcap y) \sqcup (x \sqcap z)$, and, dually, $x \sqcup (y \sqcap z) = (x \sqcup y) \sqcap (x \sqcup z)$.

In a de Morgan lattice, every element $x$ has a unique dual element $\overline{x}$, such that $\overline{x} = x$ and $x \sqsubseteq y$ implies $\overline{y} \sqsubseteq \overline{x}$.

Definition (Truth domain)

We call $\mathcal{L}$ a truth domain, if it is a finite distributive de Morgan lattice.
LTL’s semantics using truth domains

Definition (LTL semantics (common part))

Semantics of LTL formulae over a finite or infinite word \( w = a_0a_1 \ldots \in \Sigma^\infty \)

<table>
<thead>
<tr>
<th>Boolean constants</th>
<th>Boolean combinations</th>
</tr>
</thead>
<tbody>
<tr>
<td>([w \models \text{true}] \mathcal{L}) = (\top)</td>
<td>([w \models \neg \varphi] \mathcal{L}) = ([w \models \varphi] \mathcal{L})</td>
</tr>
<tr>
<td>([w \models \text{false}] \mathcal{L}) = (\bot)</td>
<td>([w \models \varphi \lor \psi] \mathcal{L}) = ([w \models \varphi] \mathcal{L} \cup [w \models \psi] \mathcal{L})</td>
</tr>
<tr>
<td>([w \models \varphi \land \psi] \mathcal{L}) = ([w \models \varphi] \mathcal{L} \cap [w \models \psi] \mathcal{L})</td>
<td></td>
</tr>
</tbody>
</table>

atomic propositions

\([w \models p] \mathcal{L}\) = \(\begin{cases} \top & \text{if } p \in a_0 \\ \bot & \text{if } p \notin a_0 \end{cases}\)

\([w \models \neg p] \mathcal{L}\) = \(\begin{cases} \top & \text{if } p \notin a_0 \\ \bot & \text{if } p \in a_0 \end{cases}\)

next X/weak next X TBD

until/release

\([w \models \varphi U \psi] \mathcal{L}\) = \(\begin{cases} \top & \text{there is a } k, 0 \leq k < |w| : [w^k \models \psi] \mathcal{L} = \top \text{ and } \\ \text{for all } l \text{ with } 0 \leq l < k : [w^l \models \varphi] = \top } \\
TBD \text{ else} \end{cases}\)

\(\varphi R \psi \equiv \neg (\neg \varphi U \neg \psi)\)
Outline

Runtime Verification

Runtime Verification for LTL

LTL over Finite, Completed Words
LTL over Finite, Non-Completed Words: Impartiality
LTL over Non-Completed Words: Anticipation
Monitorable Properties
RV-LTL
LTL with a Predictive Semantics
LTL wrap-up

Extensions

Monitoring Systems/Logging
Steering
RV frameworks
jUnit$^{RV}$ – Testing Temporal Properties

Motivating Example
jUnit$^{RV}$ – Idea
Using jUnit$^{RV}$
LTL on finite words

Application area: Specify properties of finite word
Definition (FLTL)

Semantics of FLTL formulae over a word \( u = a_0 \ldots a_{n-1} \in \Sigma^* \)

next

\[
[u \models X \varphi]_F = \begin{cases} 
[u^1 \models \varphi]_F & \text{if } u^1 \neq \epsilon \\
\bot & \text{otherwise}
\end{cases}
\]

weak next

\[
[u \models \overline{X} \varphi]_F = \begin{cases} 
[u^1 \models \varphi]_F & \text{if } u^1 \neq \epsilon \\
\top & \text{otherwise}
\end{cases}
\]
Monitoring LTL on finite words

(Bad) Idea

just compute semantics...
Outline

Runtime Verification

Runtime Verification for LTL

- LTL over Finite, Completed Words
- LTL over Finite, Non-Completed Words: Impartiality
- LTL over Non-Completed Words: Anticipation
- Monitorable Properties
- RV-LTL
- LTL with a Predictive Semantics
- LTL wrap-up

Extensions

- Monitoring Systems/Logging
- Steering
- RV frameworks
- jUnit\textsuperscript{RV} – Testing Temporal Properties
  - Motivating Example
  - jUnit\textsuperscript{RV} – Idea
  - Using jUnit\textsuperscript{RV}
LTL on finite, but not completed words

Application area: Specify properties of finite but expanding word
LTL on finite, but not completed words

Be Impartial!
- go for a final verdict (T or ⊥) only if you really know
LTL on finite, but not completed words

Be Impartial!

- go for a final verdict (\(\top\) or \(\bot\)) only if you really know
- stick to your word
LTL on finite, but not complete words

Impartiality implies multiple values

Every two-valued logic is not impartial.

Definition (FLTL$_4$)

Semantics of FLTL formulae over a word $u = a_0 \ldots a_{n-1} \in \Sigma^*$

next

$$[u \models X \varphi]_4 = \begin{cases} 
[u^1 \models \varphi]_4 & \text{if } u^1 \neq \epsilon \\
\perp^p & \text{otherwise}
\end{cases}$$

weak next

$$[u \models \bar{X} \varphi]_4 = \begin{cases} 
[u^1 \models \varphi]_4 & \text{if } u^1 \neq \epsilon \\
\top^p & \text{otherwise}
\end{cases}$$
Monitoring LTL on finite but expanding words

Left-to-right!
Monitoring LTL on finite but expanding words

Rewriting

Idea: Use rewriting of formula

Evaluating FLTL4 for each subsequent letter

- evaluate atomic propositions
- evaluate next-formulas
- that’s it thanks to
  \[ \varphi \mathrel{U}\psi \equiv \psi \lor (\varphi \land X\varphi \mathrel{U}\psi) \]
  
  and
  
  \[ \varphi \mathrel{R}\psi \equiv \psi \land (\varphi \lor \bar{X}\varphi \mathrel{R}\psi) \]
  
  and remember what to evaluate for the next letter
Evaluating FLTL4 for each subsequent letter

**Pseudo Code**

<table>
<thead>
<tr>
<th>evalFLTL4</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>((\top, \top))</td>
</tr>
<tr>
<td>false</td>
<td>((\bot, \bot))</td>
</tr>
<tr>
<td>p</td>
<td>(((p \in a), (p \in a)))</td>
</tr>
</tbody>
</table>
| \(\neg \phi\) | let \((\text{valPhi}, \text{phiRew}) = \text{evalFLTL4 } \phi \ a\)  
|           | in \((\text{valPhi}, \neg \text{phiRew})\) |
| \(\phi \lor \psi\) | let \((\text{valPhi}, \text{phiRew}) = \text{evalFLTL4 } \phi \ a\)  
|           | (valPsi, psiRew) = \text{evalFLTL4 } \psi \ a  
|           | in \((\text{valPhi} \lor \text{valPsi}, \text{phiRew} \lor \text{psiRew})\) |
| \(\phi \land \psi\) | let \((\text{valPhi}, \text{phiRew}) = \text{evalFLTL4 } \phi \ a\)  
|           | (valPsi, psiRew) = \text{evalFLTL4 } \psi \ a  
|           | in \((\text{valPhi} \land \text{valPsi}, \text{phiRew} \land \text{psiRew})\) |
| \(\phi U \psi\) | a = \text{evalFLTL4 } \psi \lor ((\phi \land X(\phi U \psi)) \ a) |
| \(\phi R \psi\) | a = \text{evalFLTL4 } \psi \land ((\phi \lor \bar{X}(\phi R \psi)) \ a) |
| \(X\phi\)   | a = \((\perp^p, \phi)\) |
| \(\bar{X}\phi\) | a = \((\top^p, \phi)\) |
Monitoring LTL on finite but expanding words

Automata-theoretic approach

- Synthesize automaton
- Monitoring = stepping through automaton
Rewriting vs. automata

Rewriting function defines transition function

| evalFLTL4 | true | a = (⊤, true) |
| evalFLTL4 | false | a = (⊥, false) |
| evalFLTL4 | p | a = ((p in a), (p in a) ? true : false) |
| evalFLTL4 | ¬φ | a = let (valPhi, phiRew) = evalFLTL4 φ a in (valPhi, ¬phiRew) |
| evalFLTL4 | ϕ ∨ ψ | a = let |
| | | (valPhi, phiRew) = evalFLTL4 φ a |
| | | (valPsi, psiRew) = evalFLTL4 ψ a |
| | | in (valPhi ⊔ valPsi, phiRew ∨ psiRew) |
| evalFLTL4 | ϕ ∧ ψ | a = let |
| | | (valPhi, phiRew) = evalFLTL4 φ a |
| | | (valPsi, psiRew) = evalFLTL4 ψ a |
| | | in (valPhi ⊓ valPsi, phiRew ∧ psiRew) |
| evalFLTL4 | ϕ U ψ | a = evalFLTL4 ψ ∨ (φ ∧ X(φ U ψ)) a |
| evalFLTL4 | ϕ R ψ | a = evalFLTL4 ψ ∧ (φ ∨ X(φ R ψ)) a |
| evalFLTL4 | Xφ | a = (⊥^p, φ) |
| evalFLTL4 | Xφ | a = (⊤^p, φ) |
Automata-theoretic approach

The roadmap

- alternating Mealy machines
Automata-theoretic approach

The roadmap

- alternating Mealy machines
- Moore machines
Automata-theoretic approach

The roadmap

- alternating Mealy machines
- Moore machines
- alternating machines
Automata-theoretic approach

The roadmap

- alternating Mealy machines
- Moore machines
- alternating machines
- non-deterministic machines
Automata-theoretic approach

The roadmap

- alternating Mealy machines
- Moore machines
- alternating machines
- non-deterministic machines
- deterministic machines
Automata-theoretic approach

The roadmap

- alternating Mealy machines
- Moore machines
- alternating machines
- non-deterministic machines
- deterministic machines
- state sequence for an input word
Definition (Alternating Mealy Machine)

A alternating Mealy machine is a tupel \( \mathcal{M} = (Q, \Sigma, \Gamma, q_0, \delta) \) where

- \( Q \) is a finite set of states,
- \( \Sigma \) is the input alphabet,
- \( \Gamma \) is a finite, distributive lattice, the output lattice,
- \( q_0 \in Q \) is the initial state and
- \( \delta : Q \times \Sigma \rightarrow B^+(\Gamma \times Q) \) is the transition function
Supporting alternating finite-state machines

Definition (Alternating Mealy Machine)

A alternating Mealy machine is a tupel $\mathcal{M} = (Q, \Sigma, \Gamma, q_0, \delta)$ where

- $Q$ is a finite set of states,
- $\Sigma$ is the input alphabet,
- $\Gamma$ is a finite, distributive lattice, the output lattice,
- $q_0 \in Q$ is the initial state and
- $\delta : Q \times \Sigma \rightarrow B^+ (\Gamma \times Q)$ is the transition function.

Convention

Understand $\delta : Q \times \Sigma \rightarrow B^+ (\Gamma \times Q)$ as a function $\delta : Q \times \Sigma \rightarrow \Gamma \times B^+ (Q)$. 
A run of an alternating Mealy machine $M = (Q, \Sigma, \Gamma, q_0, \delta)$ on a finite word $u = a_0 \ldots a_{n-1} \in \Sigma^+$ is a sequence $t_0 \rightarrow t_1 \rightarrow \ldots \rightarrow t_{n-1} \rightarrow t_n$ such that

- $t_0 = q_0$ and
- $(t_i, b_{i-1}) = \hat{\delta}(t_{i-1}, a_{i-1})$

where $\hat{\delta}$ is inductively defined as follows

- $\hat{\delta}(q, a) = \delta(q, a)$,
- $\hat{\delta}(q \lor q', a) = (\hat{\delta}(q, a)|_1 \sqcup \hat{\delta}(q', a)|_1, \hat{\delta}(q, a)|_2 \lor \hat{\delta}(q', a)|_2)$, and
- $\hat{\delta}(q \land q', a) = (\hat{\delta}(q, a)|_1 \sqcap \hat{\delta}(q', a)|_1, \hat{\delta}(q, a)|_2 \land \hat{\delta}(q', a)|_2)$

The output of the run is $b_{n-1}$. 
Transition function of an alternating Mealy machine

Transition function $\delta^a_4 : Q \times \Sigma \rightarrow B^+(\Gamma \times Q)$

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\delta^a_4(\text{true}, a)$</td>
<td>$(\top, \text{true})$</td>
</tr>
<tr>
<td>$\delta^a_4(\text{false}, a)$</td>
<td>$(\bot, \text{false})$</td>
</tr>
<tr>
<td>$\delta^a_4(p, a)$</td>
<td>$(p \in a, [p \in a])$</td>
</tr>
<tr>
<td>$\delta^a_4(\varphi \lor \psi, a)$</td>
<td>$\delta^a_4(\varphi, a) \lor \delta^a_4(\psi, a)$</td>
</tr>
<tr>
<td>$\delta^a_4(\varphi \land \psi, a)$</td>
<td>$\delta^a_4(\varphi, a) \land \delta^a_4(\psi, a)$</td>
</tr>
<tr>
<td>$\delta^a_4(\varphi \cup \psi, a)$</td>
<td>$\delta^a_4(\psi \lor (\varphi \land X(\varphi \cup \psi)), a)$</td>
</tr>
<tr>
<td></td>
<td>$= \delta^a_4(\psi, a) \lor (\delta^a_4(\varphi, a) \land (\varphi \cup \psi))$</td>
</tr>
<tr>
<td>$\delta^a_4(\varphi \cdot \psi, a)$</td>
<td>$\delta^a_4(\psi \land (\varphi \lor X(\varphi \cdot \psi)), a)$</td>
</tr>
<tr>
<td></td>
<td>$= \delta^a_4(\psi, a) \land (\delta^a_4(\varphi, a) \lor (\varphi \cdot \psi))$</td>
</tr>
<tr>
<td>$\delta^a_4(X\varphi, a)$</td>
<td>$(\bot^p, \varphi)$</td>
</tr>
<tr>
<td>$\delta^a_4(\overline{X}\varphi, a)$</td>
<td>$(\top^p, \varphi)$</td>
</tr>
</tbody>
</table>
Outline

Runtime Verification

Runtime Verification for LTL

LTL over Finite, Completed Words
LTL over Finite, Non-Completed Words: Impartiality
LTL over Non-Completed Words: Anticipation
Monitorable Properties
RV-LTL
LTL with a Predictive Semantics
LTL wrap-up

Extensions

Monitoring Systems/Logging
Steering
RV frameworks

jUnit$^{RV}$ – Testing Temporal Properties
Motivating Example
jUnit$^{RV}$ – Idea
Using jUnit$^{RV}$
Consider possible extensions of the non-completed word
Basic idea

- LTL over infinite words is commonly used for specifying correctness properties.
- Finite words in RV: prefixes of infinite, so-far unknown words.
- Re-use existing semantics.
Basic idea

- LTL over infinite words is commonly used for specifying correctness properties
- finite words in RV: prefixes of infinite, so-far unknown words
- re-use existing semantics

3-valued semantics for LTL over finite words

\[ [u \models \varphi] = \begin{cases} \top & \text{if } \forall \sigma \in \Sigma^\omega : u\sigma \models \varphi \\ \bot & \text{if } \forall \sigma \in \Sigma^\omega : u\sigma \not\models \varphi \\ ? & \text{else} \end{cases} \]
Impartial Anticipation

Impartial

- Stay with $\top$ and $\bot$
Impartial Anticipation

**Impartial**
- Stay with $\top$ and $\bot$

**Anticipatory**
- Go for $\top$ or $\bot$
- Consider $XXXfalse$

$$\epsilon \models XXXfalse$$
Impartial

- Stay with \( \top \) and \( \bot \)

Anticipatory

- Go for \( \top \) or \( \bot \)
- Consider \( XXXfalse \)

\[
\begin{align*}
\epsilon & \models XXXfalse \\
a & \models XXfalse
\end{align*}
\]
Impartial Anticipation

Impartial

▶ Stay with $\top$ and $\bot$

Anticipatory

▶ Go for $\top$ or $\bot$

▶ Consider $XXXfalse$

\[
\begin{align*}
\epsilon & \models XXXfalse \\
a & \models XXfalse \\
aa & \models Xfalse
\end{align*}
\]
Impartial Anticipation

**Impartial**

- Stay with $\top$ and $\bot$

**Anticipatory**

- Go for $\top$ or $\bot$
- Consider $XXXfalse$

\[
\begin{align*}
\epsilon & \models XXXfalse \\
ad & \models XXfalse \\
\text{aaa} & \models Xfalse \\
\text{aaa} & \models false \\
\end{align*}
\]

\[
[\epsilon \models XXXfalse] = \begin{cases} 
\top & \text{if } \forall \sigma \in \Sigma^{\omega} : \epsilon \sigma \models XXXfalse \\
\bot & \text{if } \forall \sigma \in \Sigma^{\omega} : \epsilon \sigma \not\models XXXfalse \\
? & \text{else}
\end{cases}
\]
Büchi automata (BA)
Büchi automata (BA)
Büchi automata (BA)
Büchi automata (BA)
Büchi automata (BA)
Büchi automata (BA)
Büchi automata (BA)
Büchi automata (BA)
Büchi automata (BA)
Büchi automata (BA)

\[
a b a b \ldots
\]
Büchi automata (BA)

\[ (ab)^{\omega} \in \mathcal{L}(A) \]
Büchi automata (BA)

\[
\begin{align*}
(aba)^\omega & \in \mathcal{L}(A) \\
(ab)^*aa\{a, b\}^\omega & \subseteq \mathcal{L}(A)
\end{align*}
\]
Büchi automata (BA)

Emptiness test:

\[ (ab)^\omega \in \mathcal{L}(A) \]
\[ (ab)^*aa\{a,b\}^\omega \subseteq \mathcal{L}(A) \]
Büchi automata (BA)

Emptiness test: SCCC, Tarjan

\[ (ab)^\omega \in \mathcal{L}(A) \]
\[ (ab)^*aa\{a, b\}^\omega \subseteq \mathcal{L}(A) \]
Translation of an LTL formula $\varphi$ into Büchi automata $A_\varphi$ with $\mathcal{L}(A_\varphi) = \mathcal{L}(\varphi)$

Complexity: Exponential in the length of $\varphi$
Monitor construction – Idea I

\[
[u \models \varphi] = \begin{cases} 
\top & \text{if } \forall \sigma \in \Sigma^\omega : u\sigma \models \varphi \\
\bot & \text{if } \forall \sigma \in \Sigma^\omega : u\sigma \not\models \varphi \\
? & \text{else}
\end{cases}
\]
Monitor construction – Idea I

\[ [u \models \varphi] = \begin{cases} \top & \text{if } \forall \sigma \in \Sigma^\omega : u\sigma \models \varphi \\ \bot & \text{if } \forall \sigma \in \Sigma^\omega : u\sigma \not\models \varphi \\ ? & \text{else} \end{cases} \]
Monitor construction – Idea I

\[ [u \models \varphi] = \begin{cases} 
\top & \text{if } \forall \sigma \in \Sigma^\omega : u\sigma \models \varphi \\
\bot & \text{if } \forall \sigma \in \Sigma^\omega : u\sigma \not\models \varphi \\
? & \text{else}
\end{cases} \]
Monitor construction – Idea I

\[
[u \models \varphi] = \begin{cases} 
\top & \text{if } \forall \sigma \in \Sigma^\omega : u\sigma \models \varphi \\
\bot & \text{if } \forall \sigma \in \Sigma^\omega : u\sigma \not\models \varphi \\
? & \text{else}
\end{cases}
\]
monitor construction – Idea II
monitor construction – Idea II
monitor construction – Idea II
monitor construction – Idea II

NFA

\[ \mathcal{F}_\varphi : Q_\varphi \rightarrow \{ \top, \bot \} \] Emptiness per state
The complete construction

The construction

\[ \varphi \rightarrow \text{BA} \varphi \rightarrow \mathcal{F} \varphi \rightarrow \text{NFA} \varphi \]

Lemma

\[
[u \models \varphi] = \begin{cases} 
\top & \text{if } u \notin \mathcal{L}(\text{NFA} \varphi) \\
\bot & \text{otherwise}
\end{cases}
\]
The complete construction

The construction

$$\varphi \longrightarrow \text{BA}\varphi \longrightarrow \mathcal{F}\varphi \longrightarrow \text{NFA}\varphi$$

$$\neg\varphi$$

Lemma

$$[u \models \varphi] = \begin{cases} \top & \text{if } u \notin \mathcal{L}(\text{NFA}\varphi) \\ \bot & \text{if } u \notin \mathcal{L}(\text{NFA}\varphi) \\ ? & \text{if } u \notin \mathcal{L}(\text{NFA}\varphi) \end{cases}$$
The complete construction

The construction

\[\varphi \rightarrow BA\varphi \rightarrow F\varphi \rightarrow NFA\varphi\]

\[\neg\varphi \rightarrow BA\neg\varphi \rightarrow F\neg\varphi \rightarrow NFA\neg\varphi\]

Lemma

\([u \models \varphi] = \begin{cases} 
\top & \text{if } u \notin L(NFA\neg\varphi) \\
\bot & \text{if } u \notin L(NFA\varphi) \\
? & \text{else}
\end{cases}\]
The complete construction

The construction

\[ \varphi \rightarrow \text{BA}^\varphi \rightarrow \mathcal{F}^\varphi \rightarrow \text{NFA}^\varphi \]

\[ \neg \varphi \rightarrow \text{BA}^{\neg \varphi} \rightarrow \mathcal{F}^{\neg \varphi} \rightarrow \text{NFA}^{\neg \varphi} \]
The complete construction

\[ \varphi \rightarrow \text{BA}^\varphi \rightarrow \mathcal{F}^\varphi \rightarrow \text{NFA}^\varphi \rightarrow \text{DFA}^\varphi \]

\[ \neg \varphi \rightarrow \text{BA}^{\neg \varphi} \rightarrow \mathcal{F}^{\neg \varphi} \rightarrow \text{NFA}^{\neg \varphi} \rightarrow \text{DFA}^{\neg \varphi} \]
The complete construction

The construction

\( \varphi \rightarrow \text{BA} \varphi \rightarrow \text{F} \varphi \rightarrow \text{NFA} \varphi \rightarrow \text{DFA} \varphi \)

\( \neg \varphi \rightarrow \text{BA} \neg \varphi \rightarrow \neg \text{F} \varphi \rightarrow \neg \text{NFA} \varphi \rightarrow \neg \text{DFA} \varphi \)
Static initialisation order fiasco

\[
\neg \text{spawnUinit} \quad \neg (\neg \text{spawnUinit})
\]
Static initialisation order fiasco

\[ \neg \text{spawnUinit} \quad \text{or} \quad \neg(\neg \text{spawnUinit}) \]

↓

↓
Static initialisation order fiasco

\[\neg\text{spawnUinit} \quad \quad \neg(\neg\text{spawnUinit})\]

\[\neg\text{spawn} \quad \text{true} \quad \quad \neg\text{init} \quad \text{true} \quad \text{spawn} \land \neg\text{init}\]
Static initialisation order fiasco

\( \neg \text{spawnUinit} \)

\[ \downarrow \downarrow \]

\( \neg \text{spawn} \quad \text{true} \)

\( \neg \text{init} \quad \text{true} \)

\( \neg (\neg \text{spawnUinit}) \)

\[ \downarrow \downarrow \]

\( \neg \text{init} \quad \text{spawn} \land \neg \text{init} \)

\( \text{true} \)
Static initialisation order fiasco

\( \neg \text{spawnUinit} \)

\( \Rightarrow \)

\( \neg \text{spawn} \quad \text{true} \)

\( \Rightarrow \)

\( \neg \text{spawn} \quad \text{true} \)

\( \Rightarrow \)

\( \text{true} \quad \text{true} \)

\( \Rightarrow \)

\( \text{true} \quad \text{true} \)

\( \Rightarrow \)

\( \neg \text{spawnUinit} \)

\( \neg (\neg \text{spawnUinit}) \)

\( \Rightarrow \)

\( \neg \text{init} \quad \text{true} \)

\( \Rightarrow \)

\( \text{true} \quad \neg \text{init} \)

\( \Rightarrow \)

\( \text{true} \quad \text{true} \)

\( \Rightarrow \)

\( \text{true} \quad \text{true} \)

\( \Rightarrow \)

\( \text{true} \quad \text{true} \)
Static initialisation order fiasco

\[ \neg \text{spawnUinit} \]
\[ \neg (\neg \text{spawnUinit}) \]

\[ \neg \text{spawn} \quad \text{true} \]
\[ \neg \text{init} \quad \text{true} \]
\[ \text{spawn} \quad \text{init} \]
\[ \text{spawn} \quad \text{init} \]
Static initialisation order fiasco

\[ \neg \text{spawnUinit} \]
\[ \downarrow \downarrow \]
\[ \neg \text{spawn} \]
\[ \text{true} \]
\[ \text{init} \]
\[ \neg \text{spawn} \]
\[ \text{true} \]
\[ \text{init} \]
\[ \neg \text{spawn} \]
\[ \text{true} \]
\[ \text{init} \]
\[ \neg \text{spawn} \]
\[ \text{true} \]
\[ \text{init} \]
\[ \neg \text{spawn} \]
\[ \text{true} \]
\[ \text{init} \]
\[ \neg \text{spawn} \]
\[ \text{true} \]
\[ \text{init} \]
\[ \neg \text{spawn} \]
\[ \text{true} \]
\[ \text{init} \]
\[ \neg \text{spawn} \]
\[ \text{true} \]
\[ \text{init} \]
\[ \neg \text{spawn} \]
\[ \text{true} \]
\[ \text{init} \]
\[ \neg \text{spawn} \]
\[ \text{true} \]
\[ \text{init} \]
\[ \neg \text{spawn} \]
\[ \text{true} \]
\[ \text{init} \]
\[ \neg \text{spawn} \]
\[ \text{true} \]
\[ \text{init} \]
\[ \neg \text{spawn} \]
\[ \text{true} \]
\[ \text{init} \]
\[ \neg \text{spawn} \]
\[ \text{true} \]
\[ \text{init} \]
\[ \neg \text{spawn} \]
\[ \text{true} \]
\[ \text{init} \]
\[ \neg \text{spawn} \]
\[ \text{true} \]
\[ \text{init} \]
\[ \neg \text{spawn} \]
\[ \text{true} \]
\[ \text{init} \]
\[ \neg \text{spawn} \]
\[ \text{true} \]
\[ \text{init} \]
\[ \neg \text{spawn} \]
\[ \text{true} \]
\[ \text{init} \]
\[ \neg \text{spawn} \]
\[ \text{true} \]
\[ \text{init} \]
\[ \neg \text{spawn} \]
\[ \text{true} \]
\[ \text{init} \]
Complexity

The construction

\[ \varphi \rightarrow \text{BA} \varphi \rightarrow \mathcal{F} \varphi \rightarrow \text{NFA} \varphi \rightarrow \text{DFA} \varphi \]

\[ \neg \varphi \rightarrow \text{BA} \neg \varphi \rightarrow \mathcal{F} \neg \varphi \rightarrow \text{NFA} \neg \varphi \rightarrow \text{DFA} \neg \varphi \]
Complexity

The construction

\[ \varphi \rightarrow \text{BA}^\varphi \rightarrow \mathcal{F}^\varphi \rightarrow \text{NFA}^\varphi \rightarrow \text{DFA}^\varphi \]

\[ \neg \varphi \rightarrow \text{BA}^{\neg \varphi} \rightarrow \mathcal{F}^{\neg \varphi} \rightarrow \text{NFA}^{\neg \varphi} \rightarrow \text{DFA}^{\neg \varphi} \]
The construction

\[ \varphi \rightarrow BA \rightarrow \mathcal{F} \rightarrow NFA \rightarrow DFA \rightarrow M \]
Complexity

The construction

\[ \phi \rightarrow \text{BA}^\phi \rightarrow \mathcal{F}^\phi \rightarrow \text{NFA}^\phi \rightarrow \text{DFA}^\phi \rightarrow M \]

Complexity

\[ |M| \leq 2^{2^{|\phi|}} \]
Complexity

The construction

\[ \varphi \rightarrow \mathcal{BA}^\varphi \rightarrow \mathcal{F}^\varphi \rightarrow \mathcal{NFA}^\varphi \rightarrow \mathcal{DFA}^\varphi \rightarrow M \]

\[ |M| \leq 2^{2|\varphi|} \]

Optimal result!

FSM can be minimised (Myhill-Nerode)
On-the-fly Construction

The construction

\[ \phi \rightarrow BA \phi \rightarrow \mathcal{F} \phi \rightarrow NFA \phi \rightarrow DFA \phi \]

\[ \neg \phi \rightarrow BA \neg \phi \rightarrow \mathcal{F} \neg \phi \rightarrow NFA \neg \phi \rightarrow DFA \neg \phi \]
Outline

Runtime Verification

Runtime Verification for LTL

- LTL over Finite, Completed Words
- LTL over Finite, Non-Completed Words: Impartiality
- LTL over Non-Completed Words: Anticipation

Monitorable Properties

- RV-LTL
- LTL with a Predictive Semantics
- LTL wrap-up

Extensions

- Monitoring Systems/Logging
- Steering
- RV frameworks

jUnit$^{RV}$ – Testing Temporal Properties

- Motivating Example
- jUnit$^{RV}$ – Idea
- Using jUnit$^{RV}$
Monitorability

When does anticipation help?

![Mountain Landscape](image-url)
Monitors revisited

Structure of Monitors

```
bad
“⊥”
T
```

```
ugly
“?”
T
```

```
good
“⊤”
T
```
Monitors revisited

Structure of Monitors

Classification of Prefixes of Words

- Bad prefixes

[Kupferman & Vardi’01]
Monitors revisited

Structure of Monitors

Classification of Prefixes of Words

- Bad prefixes

[Kupferman & Vardi’01]
Monitors revisited

Structure of Monitors

Classification of Prefixes of Words

- Bad prefixes
- Good prefixes

[Kupferman & Vardi’01]

[Kupferman & Vardi’01]
Monitors revisited

Structure of Monitors

Classification of Prefixes of Words

- **Bad prefixes**
  - [Kupferman & Vardi’01]

- **Good prefixes**
  - [Kupferman & Vardi’01]
Monitors revisited

Structure of Monitors

Classification of Prefixes of Words

- **Bad prefixes**
  - [Kupferman & Vardi’01]
- **Good prefixes**
  - [Kupferman & Vardi’01]
- **Ugly prefixes**
Monitors revisited

Structure of Monitors

Classification of Prefixes of Words

- Bad prefixes
- Good prefixes
- Ugly prefixes

[Kupferman & Vardi’01]

[Kupferman & Vardi’01]
### Monitorable

#### Non-Monitorable [Pnueli & Zaks’07]

$\varphi$ is **non-monitorable after** $u$, if $u$ cannot be extended to a bad oder good prefix.

#### Monitorable

$\varphi$ is **monitorable** if there is no such $u$. 
Monitorable Non-Monitorable [Pnueli & Zaks’07]

\( \varphi \) is non-monitorable after \( u \), if \( u \) cannot be extended to a bad oder good prefix.

Monitorable

\( \varphi \) is monitorable if there is no such \( u \).
Monitorable Properties

Safety Properties
Monitorable Properties

Safety Properties
Monitorable Properties

Safety Properties
Monitorable Properties

Safety Properties

Co-Safety Properties
Monitorable Properties

Safety Properties

Co-Safety Properties
Monitorable Properties

Safety Properties

Co-Safety Properties
Monitorable Properties

Safety Properties

Co-Safety Properties

Note
Safety and Co-Safety Properties are monitorable
Safety- and Co-Safety-Properties

The class of monitorable properties comprises safety- and co-safety properties, but is strictly larger than their union.

Proof

Consider \(((p \lor q) Ur) \lor Gp\)
Outline

Runtime Verification

Runtime Verification for LTL
- LTL over Finite, Completed Words
- LTL over Finite, Non-Completed Words: Impartiality
- LTL over Non-Completed Words: Anticipation
- Monitorable Properties
  - RV-LTL
  - LTL with a Predictive Semantics
  - LTL wrap-up

Extensions

Monitoring Systems/Logging

Steering

RV frameworks

JUnit<sup>RV</sup> – Testing Temporal Properties
  - Motivating Example
  - jUnit<sup>RV</sup> – Idea
  - Using jUnit<sup>RV</sup>
Basic idea

- Use $\text{LTL}_3$ for $\top$ and $\bot$, use $\text{FLTL}_4$ or $\text{FLTL}$ to refine?
RV-LTL

Basic idea

- Use LTL$_3$ for $\top$ and $\bot$, use FLTL$_4$ or FLTL to refine?

4-valued semantics for LTL over finite words

$$[u \models \varphi]_{RV} = \begin{cases} 
\top & \text{if } [u \models \varphi]_3 = \top \\
\bot & \text{if } [u \models \varphi]_3 = \bot \\
\top^p & \text{if } [u \models \varphi]_3 = ? \text{ and } [u \models \varphi]_4 = \top^p \\
\bot^p & \text{if } [u \models \varphi]_3 = ? \text{ and } [u \models \varphi]_4 = \bot^p 
\end{cases}$$

Monitor: Combine corresponding Moore and Mealy machines...
Outline

Runtime Verification

Runtime Verification for LTL
  LTL over Finite, Completed Words
  LTL over Finite, Non-Completed Words: Impartiality
  LTL over Non-Completed Words: Anticipation
  Monitorable Properties
  RV-LTL
    LTL with a Predictive Semantics
    LTL wrap-up

Extensions
  Monitoring Systems/Logging
  Steering
  RV frameworks
  jUnit$^{\text{RV}}$ – Testing Temporal Properties
    Motivating Example
    jUnit$^{\text{RV}}$ – Idea
    Using jUnit$^{\text{RV}}$
Fusing model checking and runtime verification

LTL with a predictive semantics
Recall anticipatory LTL semantics

The truth value of a LTL$_3$ formula $\varphi$ wrt. $u$, denoted by $[u \models \varphi]$, is an element of $\mathbb{B}_3$ defined by

$$[u \models \varphi] = \begin{cases} \top & \text{if } \forall \sigma \in \Sigma^\omega : u\sigma \models \varphi \\ \bot & \text{if } \forall \sigma \in \Sigma^\omega : u\sigma \not\models \varphi \\ ? & \text{otherwise.} \end{cases}$$
Assumptions about environment

Definition (Semantics of LTL with Assumptions)

Let $\hat{P}$ be an assumption on possible runs of the underlying system. Let $u \in \Sigma^*$ denote a finite trace. The *truth value* of $u$ and an LTL$_3$ formula $\varphi$ wrt. $\hat{P}$, denoted by $[u \models_{\hat{P}} \varphi]$, is an element of $\mathbb{B}_3 \cup \{?, \top\}$ and defined as follows:

$$[u \models_{\hat{P}} \varphi] = \begin{cases} 
? & u \notin \omega \hat{P}, \text{ else,} \\
\top & \text{if } \forall \sigma \in \Sigma^\omega \text{ with } u\sigma \in \hat{P} : u\sigma \models \varphi \\
\bot & \text{if } \forall \sigma \in \Sigma^\omega \text{ with } u\sigma \in \hat{P} : u\sigma \not\models \varphi \\
? & \text{else} 
\end{cases}$$
Assuming program is known, applied to the empty word

Empty word $\epsilon$

$[\epsilon \models \varphi]_P = T$

iff $\forall \sigma \in \Sigma^\omega$ with $\epsilon \sigma \in P : \epsilon \sigma \models \varphi$

iff $L(P) \models \varphi$

RV more difficult than MC?

Then runtime verification implicitly answers model checking
An over-abstraction or and over-approximation of a program $\mathcal{P}$ is a program $\hat{\mathcal{P}}$ such that $\mathcal{L}(\mathcal{P}) \subseteq \mathcal{L}(\hat{\mathcal{P}}) \subseteq \Sigma^\omega$. 
Predictive Semantics

Definition (Predictive semantics of LTL)

Let $P$ be a program and let $\hat{P}$ be an over-approximation of $P$. Let $u \in \Sigma^*$ denote a finite trace. The truth value of $u$ and an LTL$_3$ formula $\varphi$ wrt. $\hat{P}$, denoted by $[u \models_{\hat{P}} \varphi]$, is an element of $\mathbb{B}_3$ and defined as follows:

$$[u \models_{\hat{P}} \varphi] = \begin{cases} \text{i} & \text{if } u \notin \omega \hat{P}, \text{ else,} \\ \top & \text{if } \forall \sigma \in \Sigma^\omega \text{ with } u\sigma \in \hat{P} : u\sigma \models \varphi \\ \bot & \text{if } \forall \sigma \in \Sigma^\omega \text{ with } u\sigma \in \hat{P} : u\sigma \not\models \varphi \\ \? & \text{else} \end{cases}$$

We write LTL$_P$ whenever we consider LTL formulas with a predictive semantics.
Properties of Predictive Semantics

Let $\hat{P}$ be an over-approximation of a program $P$ over $\Sigma$, $u \in \Sigma^*$, and $\varphi \in LTL$.

- Model checking is more precise than RV with the predictive semantics:
  \[ P \models \varphi \text{ implies } [u \models_{\hat{P}} \varphi] \in \{\top, ?\} \]

- RV has no false negatives: $[u \models_{\hat{P}} \varphi] = \bot$ implies $P \not\models \varphi$

- The predictive semantics of an LTL formula is more precise than LTL$_3$:
  \[ [u \models \varphi] = \top \text{ implies } [u \models_{\hat{P}} \varphi] = \top \]
  \[ [u \models \varphi] = \bot \text{ implies } [u \models_{\hat{P}} \varphi] = \bot \]

The reverse directions are in general not true.
The procedure for getting \([u \models \hat{P} \varphi]\) for a given \(\varphi\) and over-approximation \(\hat{P}\)
Outline

Runtime Verification

Runtime Verification for LTL

- LTL over Finite, Completed Words
- LTL over Finite, Non-Completed Words: Impartiality
- LTL over Non-Completed Words: Anticipation
- Monitorable Properties
- RV-LTL
- LTL with a Predictive Semantics

LTL wrap-up

Extensions

Monitoring Systems/Logging

Steering

RV frameworks

JUnit$^\text{RV}$ – Testing Temporal Properties

- Motivating Example
- jUnit$^\text{RV}$ – Idea
- Using jUnit$^\text{RV}$
Intermediate Summary

Semantics
- completed traces
  - two valued semantics
- non-completed traces
  - Impartiality
    - at least three values
  - Anticipation
    - finite traces
    - infinite traces
    - ... monitorability
- Prediction

Monitors
- left-to-right
- time versus space trade-off
  - rewriting
  - alternating automata
  - non-deterministic automata
  - deterministic automata
Presentation outline

Runtime Verification
Runtime Verification for LTL
  LTL over Finite, Completed Words
  LTL over Finite, Non-Completed Words: Impartiality
  LTL over Non-Completed Words: Anticipation
Monitorable Properties
  RV-LTL
  LTL with a Predictive Semantics
  LTL wrap-up

Extensions
  Monitoring Systems/Logging
  Steering
  RV frameworks
    jUnit$^{RV}$ – Testing Temporal Properties
      Motivating Example
      jUnit$^{RV}$ – Idea
    Using jUnit$^{RV}$
Extensions

LTL is just half of the story
## Extensions

### LTL with data

- J-LO
Extensions

LTL with data

- J-LO
- MOP (parameterized LTL)
Extensions

LTL with data

- J-LO
- MOP (parameterized LTL)
- RV for LTL with integer constraints
Extensions

LTL with data

- J-LO
- MOP (parameterized LTL)
- RV for LTL with integer constraints
Extensions

LTL with data

- J-LO
- MOP (parameterized LTL)
- RV for LTL with integer constraints

Further “rich” approaches

- LOLA
## Extensions

### LTL with data

- J-LO
- MOP (parameterized LTL)
- RV for LTL with integer constraints

### Further “rich” approaches

- LOLA
- Eagle (etc.)
Extensions

LTL with data
- J-LO
- MOP (parameterized LTL)
- RV for LTL with integer constraints

Further “rich” approaches
- LOLA
- Eagle (etc.)
Extensions

**LTL with data**
- J-LO
- MOP (parameterized LTL)
- RV for LTL with integer constraints

**Further “rich” approaches**
- LOLA
- Eagle (etc.)

**Further dimensions**
- real-time
Extensions

LTL with data

- J-LO
- MOP (parameterized LTL)
- RV for LTL with integer constraints

Further “rich” approaches

- LOLA
- Eagle (etc.)

Further dimensions

- real-time
- concurrency
## Extensions

### LTL with data
- J-LO
- MOP (parameterized LTL)
- RV for LTL with integer constraints

### Further “rich” approaches
- LOLA
- Eagle (etc.)

### Further dimensions
- real-time
- concurrency
- distribution
Presentation outline

Runtime Verification

Runtime Verification for LTL
  LTL over Finite, Completed Words
  LTL over Finite, Non-Completed Words: Impartiality
  LTL over Non-Completed Words: Anticipation

Monitorable Properties

RV-LTL

LTL with a Predictive Semantics

LTL wrap-up

Extensions

Monitoring Systems/Logging

Steering

RV frameworks

JUnit$^{RV}$ – Testing Temporal Properties
  Motivating Example

JUnit$^{RV}$ – Idea

Using jUnit$^{RV}$
Presentation outline

Runtime Verification
Runtime Verification for LTL
  LTL over Finite, Completed Words
  LTL over Finite, Non-Completed Words: Impartiality
  LTL over Non-Completed Words: Anticipation
Monitorable Properties
RV-LTL
LTL with a Predictive Semantics
LTL wrap-up

Extensions
Monitoring Systems/Logging

Steering
RV frameworks
jUnit^RV – Testing Temporal Properties
  Motivating Example
  jUnit^RV – Idea
Using jUnit^RV
Monitoring Systems/Logging: Overview

- Exception
- Monitoring results / steering
- Print
- Manual
- Automatically

Martin Leucker
VTSA, 2023
### React!

#### Runtime Verification

Observe—do not react

#### Realising dynamic systems

- self-healing systems
- adaptive systems, self-organising systems
- ...

---

[Text continues]
React!

Runtime Verification
Observe—do not react

Realising dynamic systems
- self-healing systems
- adaptive systems, self-organising systems
- ...
- use monitors for observation—then react
class Resource {
  /*@
  Where scope = class
  logic = PTLTL
  { Event authenticate: end(exec(*
    authenticate()));
    Event use: begin(exec(* access()));
    Formula : use -> <!> authenticate
  }
  @*/
  void authenticate() {...}
  void access() {...}
  ...
}
Monitor-based Runtime Reflection

Software Architecture Pattern

Safety-Critical System

Mitigation

Diagnosis

Monitoring

Logging
Presentation outline

Runtime Verification
Runtime Verification for LTL
  LTL over Finite, Completed Words
  LTL over Finite, Non-Completed Words: Impartiality
  LTL over Non-Completed Words: Anticipation
Monitorable Properties
RV-LTL
LTL with a Predictive Semantics
LTL wrap-up

Extensions
Monitoring Systems/Logging
Steering

RV frameworks

jUnit$^{RV}$ – Testing Temporal Properties
  Motivating Example
  jUnit$^{RV}$ – Idea
  Using jUnit$^{RV}$
Monitoring Systems/Logging: Overview

RV frameworks

- Eagle
- J-LO
- Larva
- LogScope
- LoLa
- MAC
- MOP
- RulerR
- Temporal Rover
- TraceContract
- TraceMatches
Presentation outline

Runtime Verification

Runtime Verification for LTL
  LTL over Finite, Completed Words
  LTL over Finite, Non-Completed Words: Impartiality
  LTL over Non-Completed Words: Anticipation

Monitorable Properties

RV-LTL

LTL with a Predictive Semantics

LTL wrap-up

Extensions

Monitoring Systems/Logging

Steering

RV frameworks

\textit{jUnit}^{RV} – Testing Temporal Properties

Motivating Example

\textit{jUnit}^{RV} – Idea

Using \textit{jUnit}^{RV}
Outline

Runtime Verification

Runtime Verification for LTL
  LTL over Finite, Completed Words
  LTL over Finite, Non-Completed Words: Impartiality
  LTL over Non-Completed Words: Anticipation
  Monitorable Properties
  RV-LTL
  LTL with a Predictive Semantics
  LTL wrap-up

Extensions
  Monitoring Systems/Logging
  Steering
  RV frameworks

jUnit\textsuperscript{RV} – Testing Temporal Properties
  Motivating Example
  jUnit\textsuperscript{RV} – Idea
Example Application

- Some application for data entry
- Connects to a server
- Data can be read, modified and committed
Example Application

- Frontend handles GUI
- Backend handles communication to the server
- Frontend and backend communicate via the following interface:

```java
public interface DataService {
    void connect(String userID) throws UnknownUserException;
    void disconnect();
    Data readData(String field);
    void modifyData(String field, Data data);
    void commit() throws CommitException;
}
```
A “simple” Test

- Frontend has to use backend *correctly*
- Data has to be committed before disconnecting

**Example**

```java
@Test
public void test1() {
    DataService service = new MyDataService("http://myserver.net");
    MyDataClient client = new MyDataClient(service);

    client.authenticate("daniel");
    client.addPatient("Mr. Smith");
    client.switchToUser("ruth");
    assertTrue(service.debug_committed()); // switching means logout
    client.getPatientFile("miller-2143-1");
    client.setPhone("miller-2143-1", "012345678");
    client.exit();
    assertTrue(service.debug_committed());
}
```
Observations

- Test inputs are *interleaved* with assertions
- Requires internal knowledge about the class under scrutiny
- Requires refactoring of interfaces between components
- Components might need additional logic to track temporal properties
- Production code is polluted by test code
- Program logic for temporal properties can be complicated

⇒ Classical unit testing is not suitable to assure temporal properties on internal interfaces
Outline

Runtime Verification
Runtime Verification for LTL
  LTL over Finite, Completed Words
  LTL over Finite, Non-Completed Words: Impartiality
  LTL over Non-Completed Words: Anticipation
Monitorable Properties
RV-LTL
LTL with a Predictive Semantics
LTL wrap-up
Extensions
Monitoring Systems/Logging
Steering
RV frameworks

JUnit<sup>RV</sup> – Testing Temporal Properties
  Motivating Example
  JUnit<sup>RV</sup> – Idea
Using JUnit<sup>RV</sup>
Main Ideas

- separate test as sequence of actions to do be carried out during test execution
- and monitor specification in FLTL$_4$
  - false can be used to abort a test immediately
  - true can be used to abort monitoring
  - true$_p$/false$_p$ determines the verdict for completed test runs
Outline

Runtime Verification
Runtime Verification for LTL
  LTL over Finite, Completed Words
  LTL over Finite, Non-Completed Words: Impartiality
  LTL over Non-Completed Words: Anticipation
Monitorable Properties
RV-LTL
LTL with a Predictive Semantics
LTL wrap-up

Extensions
Monitoring Systems/Logging
Steering
RV frameworks

JUnit$^{RV}$ – Testing Temporal Properties
  Motivating Example
  jUnit$^{RV}$ – Idea
Using jUnit$^{RV}$
Events and Propositions

- Formal runs consist of discrete steps in time
- When does a program perform a step?
- Explicitly specify events triggering time steps
- Only one event occurs at a point of time
- Propositions may be evaluated in the current state
Example (Specifying Events)

String dataService = "myPackage.DataService";
private static Event modify = called(dataService, "modify");
private static Event committed = returned(dataService, "commit");
private static Event disconnect = called(dataService, "disconnect");

Example (Specifying Propositions)

private static Proposition auth
    = new Proposition(eq(invoke($this, "getStatus"), AUTH);
Temporal Assertion

- LTL is used to specify temporal properties
- Generated monitors only observe the specified events
- $G(modify \rightarrow \neg disconnect \ U committed)$

Example (Specifying Monitors)

```java
private static Monitor commitBeforeDisconnect = new FLTL4Monitor(
    Always (implies (modify,
        Until (not (disconnect), committed)
    )));
```
@Test
@Monitors({"commitBeforeDisconnect"})
public void test1() {
    DataService service = new MyDataService("http://myserver.net");
    MyDataClient client = new MyDataClient(service);
    client.authenticate("daniel");
    client.addPatient("Mr. Smith");
    client.switchToUser("ruth");
    client.getPatientFile("miller-2143-1");
    client.setPhone("miller-2143-1", "012345678");
    client.exit();
}
@RunWith(RVRunner.class)
public class MyDataClientTest {

    private static final String dataServiceQname = "junitrvexamples.DataService";
    private static Event modify = called(dataServiceQname, "modifyData");
    private static Event committed = returned(dataServiceQname, "commit");
    private static Event disconnect = invoke(dataServiceQname, "disconnect");

    // create a monitor for LTL4 property G(modify -> !close U commit)
    private static Monitor commitBeforeClose = new FLTL4Monitor(
            Always(
                    implies(
                            modify,
                            Until(not(disconnect), committed)))));

    @Test
    @Monitors({"commitBeforeClose", "authWhenModify"})
    public void test1() {
        ...
    }
}
Outline

Runtime Verification

Runtime Verification for LTL
  LTL over Finite, Completed Words
  LTL over Finite, Non-Completed Words: Impartiality
  LTL over Non-Completed Words: Anticipation
  Monitorable Properties
  RV-LTL
  LTL with a Predictive Semantics
  LTL wrap-up

Extensions

Monitoring Systems/Logging

Steering

RV frameworks

JUnit$^{RV}$ – Testing Temporal Properties
  Motivating Example
  jUnit$^{RV}$ – Idea
  Using jUnit$^{RV}$
Architecture

Program

JUnit

JUnitTest

JUnitRV

LTL4RVLib

EventInjection

Javassist

...
JUnit uses test runners to execute tests
- jUnit provides a default implementation
- jUnit\textsuperscript{RV} provides \texttt{RVRunner} extending the default implementation
- jUnit\textsuperscript{RV} provides a custom \texttt{ClassLoader}
- Class loading by program under scrutiny is intercepted
- Bytecode is manipulated to intercept events
Features

- jUnit\textsuperscript{RV} is provided as single class jar file that has to be made available on the Java class path
- It can easily integrated into build systems and IDEs
- It may be used to test third party components where no byte code is available
- It may be extended with custom specification formalisms
- Test failures are reported as soon as a monitor fails
- Stack traces show the exact location of the failure in the program under scrutiny
JUnitRV Running in Netbeans
JUnitRV – Summary

- Unit testing and runtime verification are combined
- JUnit is extended by temporal assertions
- Testing temporal properties is less cumbersome
- JUnitRV integrates easily in existing projects and environments
Presentation outline

Runtime Verification

Runtime Verification for LTL
  LTL over Finite, Completed Words
  LTL over Finite, Non-Completed Words: Impartiality
  LTL over Non-Completed Words: Anticipation

Monitorable Properties

RV-LTL

LTL with a Predictive Semantics

LTL wrap-up

Extensions

Monitoring Systems/Logging

Steering

RV frameworks

jUnit$^{RV}$ – Testing Temporal Properties
  Motivating Example
  jUnit$^{RV}$ – Idea
  Using jUnit$^{RV}$
Summary

- RV needs similar temporal logics as model checking, but adaptations for
  - finite runs
  - impartiality
  - anticipation
  - prediction
- Application jUnit$^{RV}$
That’s it!

Thanks! - Questions?