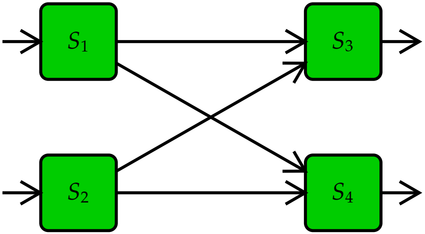
Runtime Verification

Martin Leucker

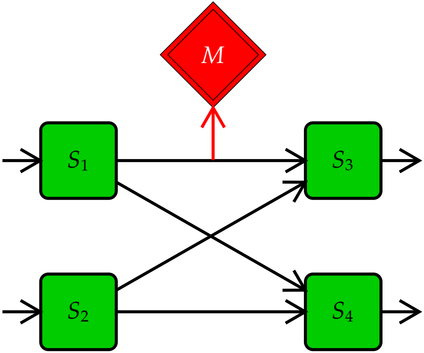
Institute for Software Engineering
Universität zu Lübeck

VTSA 2023 - Runtime Verification

Runtime Verification (RV)

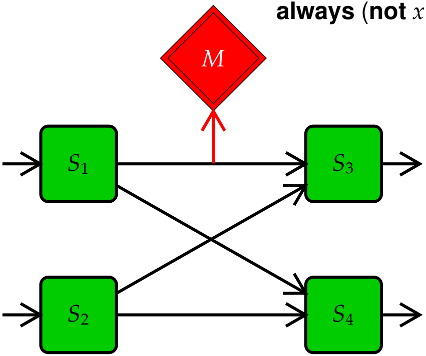


Runtime Verification (RV)

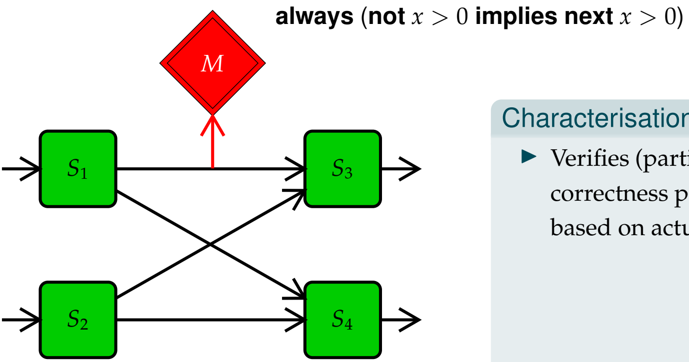


Runtime Verification (RV)

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



Runtime Verification (RV)

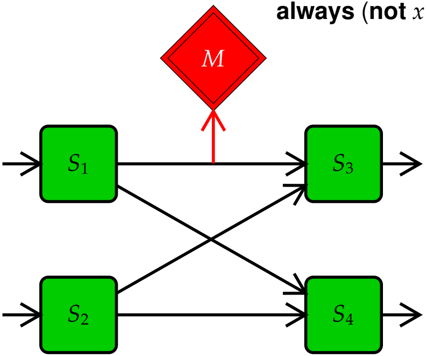


Characterisation

- Verifies (partially) correctness properties based on actual executions

Runtime Verification (RV)

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

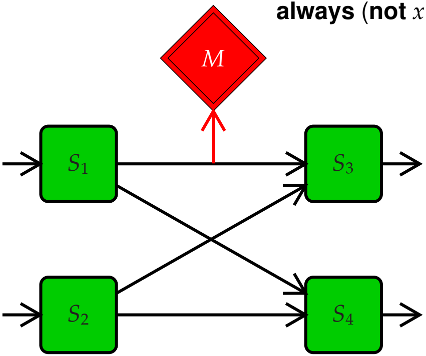


Characterisation

- Verifies (partially) correctness properties based on actual executions
- Simple verification technique

Runtime Verification (RV)

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

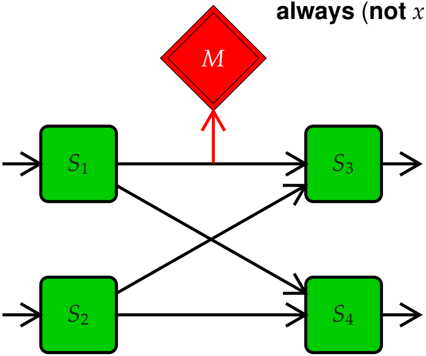


Characterisation

- Verifies (partially) correctness properties based on actual executions
- Simple verification technique
- Complementing

Runtime Verification (RV)

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

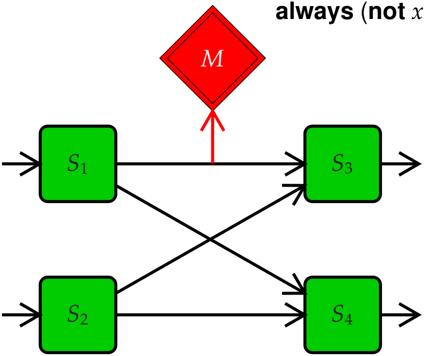


Characterisation

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

Runtime Verification (RV)

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

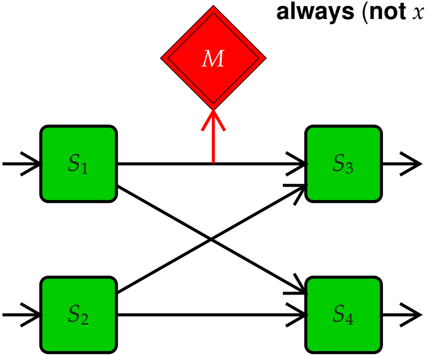


Characterisation

- 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: $w \in \mathcal{L}(\varphi)$

Model Checking

- Specification of System

Model Checking

- Specification of System
  - as formula φ of linear-time temporal logic (LTL)

Model Checking

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

Model Checking

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

Model Checking

- Specification of System
  - as formula φ 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 φ 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

Model Checking

- Specification of System
  - as formula φ 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

Testing: Input/Output Sequence

- incomplete verification technique

Testing

Testing: Input/Output Sequence

- incomplete verification technique
- test case: finite sequence of input/output actions

Testing

Testing: Input/Output Sequence

- incomplete verification technique
- test case: finite sequence of input/output actions
- test suite: finite set of test cases

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

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

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

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

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

Presentation outline

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.

Taxonomy


runtime
verification
trace
finite
finite non-
completed
infinite
integration
inline
outline
stage
online
offline
interference
invasive
non-invasive
steering
passive
active
monitoring
event
sequence
state se-
quence
input/
output
behavior
application
area
information
collection
performance
evaluation
security
safety
checking

Presentation outline

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

Runtime Verification for LTL

Definition (Syntax of LTL formulae)

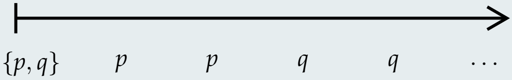
Let p be an atomic proposition from a finite set of atomic propositions AP. The set of LTL formulae, denoted with LTL, is inductively defined by the following grammar:

$$\begin{array}{rcl}\varphi & ::= & true \mid p \mid \varphi \vee \varphi \mid \varphi \; U \; \varphi \mid X\varphi \mid \\ & & false \mid \neg p \mid \varphi \wedge \varphi \mid \varphi \; R \; \varphi \mid \bar{X}\varphi \mid \\ & & \neg\varphi\end{array}$$

Linear-time Temporal Logic (LTL)

Semantics

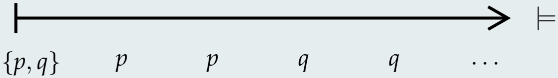
over $w \in (2^{AP})^\omega = \Sigma^\omega$



Linear-time Temporal Logic (LTL)

Semantics

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



Linear-time Temporal Logic (LTL)

Semantics

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

$\{p, q\} \quad p \quad p \quad q \quad q \quad \dots \quad \models \quad p$

$\neg p$

pUq

$X(pUq)$

Linear-time Temporal Logic (LTL)

Semantics

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

							p	√
							$\neg p$	
⟼						$\models$	pUq	
$\{p, q\}$	p	p	q	q	$\dots$		$X(pUq)$	

Linear-time Temporal Logic (LTL)

Semantics

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

$\{p,q\} \quad p \quad p \quad q \quad q \quad \dots \models$

p	$\surd$
$\neg p$	X
pUq	
$X(pUq)$	

Linear-time Temporal Logic (LTL)

Semantics

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

$\{p, q\} \quad p \quad p \quad q \quad q \quad \ldots \models$

p	√
$\neg p$	X
pUq	√
$X(pUq)$	

Linear-time Temporal Logic (LTL)

Semantics

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

$\{p, q\} \quad p \quad p \quad q \quad q \quad \dots \models$

p	✓
$\neg p$	X
pUq	✓
$X(pUq)$	✓

Linear-time Temporal Logic (LTL)

Semantics

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

$\{p, q\} \quad p \quad p \quad q \quad q \quad \cdots \quad \models$

p	√
$\neg p$	X
pUq	√
$X(pUq)$	√

Abbreviation

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

Linear-time Temporal Logic (LTL)

Semantics

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

$\{p, q\} \quad p \quad p \quad q \quad q \quad \ldots$

$\models$

p	✓
$\neg p$	X
pUq	✓
$X(pUq)$	✓

Abbreviation

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

Example

$G\neg(critic_1 \wedge critic_2), G(\neg alive \rightarrow Xalive)$

LTL on infinite words

Definition (LTL semantics (traditional))

Semantics of LTL formulae over an infinite word $w = a_0 a_1 \ldots \in \Sigma^\omega$, where $w^i = a_i a_{i+1} \ldots$

$w \models true$		
$w \models p$	if	$p \in a_0$
$w \models \neg p$	if	$p \notin a_0$
$w \models \neg\varphi$	if	not $w \models \varphi$
$w \models \varphi \vee \psi$	if	$w \models \varphi$ or $w \models \psi$
$w \models \varphi \wedge \psi$	if	$w \models \varphi$ and $w \models \psi$
$w \models X\varphi$	if	$w^1 \models \varphi$
$w \models \bar{X}\varphi$	if	$w^1 \models \varphi$
$w \models \varphi\ U\ \psi$	if	there is k with $0 \leq k < \lvert w \rvert$: $w^k \models \psi$
		and for all l with $0 \leq l < k$ $w^l \models \varphi$
$w \models \varphi\ R\ \psi$	if	for all k with $0 \leq k < \lvert w \rvert$: ($w^k \models \psi$
		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??

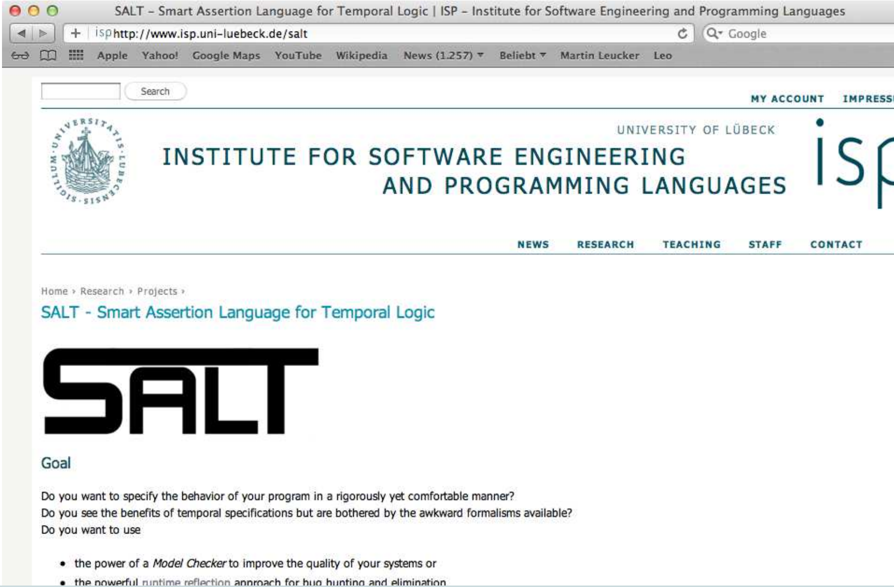
"LTL is for theoreticians—but for practitioners?"

SALT

Structured Assertion Language for Temporal Logic

"Syntactic Sugar for LTL" [Bauer, L., Streit@ICFEM'06]

SALT – `http://www.isp.uni-luebeck.de/salt`



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:

$$
\begin{array}{rcl}
\varphi & ::= & \mathit{true} \mid p \mid \varphi \vee \varphi \mid \varphi\, U\, \varphi \mid X\varphi \mid \\
& & \mathit{false} \mid \neg p \mid \varphi \wedge \varphi \mid \varphi\, R\, \varphi \mid \bar{X}\varphi \mid \\
& & \neg\varphi
\end{array}
$$

Truth Domains

Lattice

- A lattice is a partially ordered set $(\mathcal{L}, \sqsubseteq)$ where for each $x, y \in \mathcal{L}$, 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 $\mathcal{L}$ 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 $\top$.

Truth Domains (cont.)

Lattice (cont.)

- 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 $\overline{\overline{x}} = x$ 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 $w = a_0 a_1 \ldots \in \Sigma^\infty$

Boolean constants

$$[w \models true]_\mathfrak{L} = \top$$
$$[w \models false]_\mathfrak{L} = \bot$$

Boolean combinations

$$[w \models \neg\varphi]_\mathfrak{L} = \overline{[w \models \varphi]_\mathfrak{L}}$$
$$[w \models \varphi \vee \psi]_\mathfrak{L} = [w \models \varphi]_\mathfrak{L} \sqcup [w \models \psi]_\mathfrak{L}$$
$$[w \models \varphi \wedge \psi]_\mathfrak{L} = [w \models \varphi]_\mathfrak{L} \sqcap [w \models \psi]_\mathfrak{L}$$

atomic propositions

$$[w \models p]_\mathfrak{L} = \begin{cases} \top & \text{if } p \in a_0 \\ \bot & \text{if } p \notin a_0 \end{cases} \qquad [w \models \neg p]_\mathfrak{L} = \begin{cases} \top & \text{if } p \notin a_0 \\ \bot & \text{if } p \in a_0 \end{cases}$$

next X/weak next X TBD

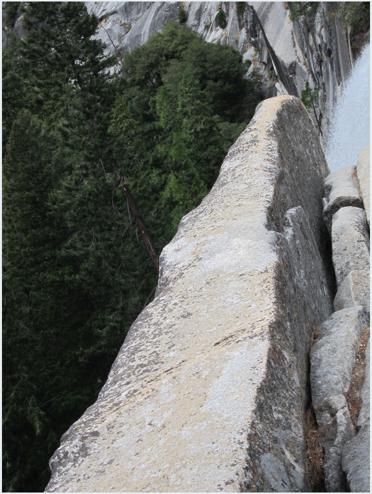
until/release

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

Outline

LTL on finite words

Application area: Specify properties of finite word



LTL on finite words

Definition (FLTL)

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

next

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

weak next

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

Monitoring LTL on finite words

(Bad) Idea

just compute semantics. . .

Outline

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 ($\top$ or $\bot$) 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 $u = a_0 \dots a_{n-1} \in \Sigma^*$

next

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

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

$$\varphi \; U \; \psi \equiv \psi \vee (\varphi \wedge X \varphi \; U \; \psi)$$

and

$$\varphi \; R \; \psi \equiv \psi \wedge (\varphi \vee \bar{X} \varphi \; R \; \psi)$$

- and remember what to evaluate for the next letter

Evaluating FLTL4 for each subsequent letter

Pseudo Code

```
evalFLTL4 true    a = (⊤,⊤)
evalFLTL4 false   a = (⊥,⊥)
evalFLTL4 p       a = ((p in a),(p in a))
evalFLTL4 ¬φ      a = let (valPhi,phiRew) = evalFLTL4 φ a
                      in (\overline{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 ∨ X̄(φ R ψ)) a
evalFLTL4 Xφ      a = (⊥^p,φ)
evalFLTL4 X̄φ      a = (⊤^p,φ)
```

Monitoring LTL on finite but expanding words

Automata-theoretic approach

- Synthesize automaton
- Monitoring = stepping through automaton

Rewriting vs. automata

Rewriting function defines transition function

$$
\begin{array}{lll}
\texttt{evalFLTL4}\ \mathit{true} & \texttt{a} = & (\top, \mathit{true}) \\
\texttt{evalFLTL4}\ \mathit{false} & \texttt{a} = & (\bot, \mathit{false}) \\
\texttt{evalFLTL4}\ \texttt{p} & \texttt{a} = & ((\texttt{p}\ \underline{\textbf{in}}\ \texttt{a}), (\texttt{p}\ \underline{\textbf{in}}\ \texttt{a})\ ?\ \mathit{true}\ :\ \mathit{false}) \\
\texttt{evalFLTL4}\ \neg\varphi & \texttt{a} = & \underline{\textbf{let}}\ (\texttt{valPhi}, \texttt{phiRew}) = \texttt{evalFLTL4}\ \varphi\ \texttt{a} \\
 & & \underline{\textbf{in}}\ (\overline{\texttt{valPhi}}, \neg\texttt{phiRew}) \\
\texttt{evalFLTL4}\ \varphi \vee \psi & \texttt{a} = & \underline{\textbf{let}} \\
 & & \quad (\texttt{valPhi}, \texttt{phiRew}) = \texttt{evalFLTL4}\ \varphi\ \texttt{a} \\
 & & \quad (\texttt{valPsi}, \texttt{psiRew}) = \texttt{evalFLTL4}\ \psi\ \texttt{a} \\
 & & \underline{\textbf{in}}\ (\texttt{valPhi} \sqcup \texttt{valPsi}, \texttt{phiRew} \vee \texttt{psiRew}) \\
\texttt{evalFLTL4}\ \varphi \wedge \psi & \texttt{a} = & \underline{\textbf{let}} \\
 & & \quad (\texttt{valPhi}, \texttt{phiRew}) = \texttt{evalFLTL4}\ \varphi\ \texttt{a} \\
 & & \quad (\texttt{valPsi}, \texttt{psiRew}) = \texttt{evalFLTL4}\ \psi\ \texttt{a} \\
 & & \underline{\textbf{in}}\ (\texttt{valPhi} \sqcap \texttt{valPsi}, \texttt{phiRew} \wedge \texttt{psiRew}) \\
\texttt{evalFLTL4}\ \varphi\ U\ \psi & \texttt{a} = & \texttt{evalFLTL4}\ \psi \vee (\varphi \wedge X(\varphi\ U\ \psi))\ \texttt{a} \\
\texttt{evalFLTL4}\ \varphi\ R\ \psi & \texttt{a} = & \texttt{evalFLTL4}\ \psi \wedge (\varphi \vee \bar{X}(\varphi\ R\ \psi))\ \texttt{a} \\
\texttt{evalFLTL4}\ X\varphi & \texttt{a} = & (\bot^p, \varphi) \\
\texttt{evalFLTL4}\ \bar{X}\varphi & \texttt{a} = & (\top^p, \varphi)
\end{array}
$$

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

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,
- Σ is the input alphabet,
- Γ is a finite, distributive lattice, the output lattice,
- $q_0 \in Q$ is the initial state and
- $\delta : Q \times \Sigma \to B^+(\Gamma \times Q)$ 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,
- Σ is the input alphabet,
- Γ is a finite, distributive lattice, the output lattice,
- $q_0 \in Q$ is the initial state and
- $\delta : Q \times \Sigma \to B^+(\Gamma \times Q)$ is the transition function

Convention

Understand $\delta : Q \times \Sigma \to B^+(\Gamma \times Q)$ as a function $\delta : Q \times \Sigma \to \Gamma \times B^+(Q)$

Supporting alternating finite-state machines

Definition (Run of an Alternating Mealy Machine)

A run of an alternating Mealy machine $\mathcal{M} = (Q, \Sigma, \Gamma, q_0, \delta)$ on a finite word $u = a_0 \ldots a_{n-1} \in \Sigma^+$ is a sequence $t_0 \overset{(a_0,b_0)}{\longrightarrow} t_1 \overset{(a_1,b_1)}{\longrightarrow} \ldots t_{n-1} \overset{(a_{n-1},b_{n-1})}{\longrightarrow} t_n$ 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

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

The output of the run is b_{n-1}.

Transition function of an alternating Mealy machine

Transition function $\delta_4^a : Q \times \Sigma \to B^+(\Gamma \times Q)$

$$
\begin{aligned}
\delta_4^a(true, a) &= (\top, true) \\
\delta_4^a(false, a) &= (\bot, false) \\
\delta_4^a(p, a) &= (p \in a, [p \in a]) \\
\delta_4^a(\varphi \vee \psi, a) &= \delta_4^a(\varphi, a) \vee \delta_4^a(\psi, a) \\
\delta_4^a(\varphi \wedge \psi, a) &= \delta_4^a(\varphi, a) \wedge \delta_4^a(\psi, a) \\
\delta_4^a(\varphi\ U\ \psi, a) &= \delta_4^a(\psi \vee (\varphi \wedge X(\varphi\ U\ \psi)), a) \\
&= \delta_4^a(\psi, a) \vee (\delta_4^a(\varphi, a) \wedge (\varphi\ U\ \psi)) \\
\delta_4^a(\varphi\ R\ \psi, a) &= \delta_4^a(\psi \wedge (\varphi \vee \bar{X}(\varphi\ R\ \psi)), a) \\
&= \delta_4^a(\psi, a) \wedge (\delta_4^a(\varphi, a) \vee (\varphi\ R\ \psi)) \\
\delta_4^a(X\varphi, a) &= (\bot^p, \varphi) \\
\delta_4^a(\bar{X}\varphi, a) &= (\top^p, \varphi)
\end{aligned}
$$

Outline

Anticipatory Semantics

Consider possible extensions of the non-completed word

LTL for RV [BLS@FSTTCS'06]

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

LTL for RV [BLS@FSTTCS'06]

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 $XXX\mathit{false}$

$$\epsilon \models XXX\mathit{false}$$

Impartial Anticipation

Impartial

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

Anticipatory

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

$$\begin{array}{ccc} \epsilon & \models & XXXfalse \\ a & \models & XXfalse \end{array}$$

Impartial Anticipation

Impartial

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

Anticipatory

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

$$\begin{array}{lcl} \epsilon & \models & XXXfalse \\ a & \models & XXfalse \\ aa & \models & Xfalse \end{array}$$

Impartial Anticipation

Impartial

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

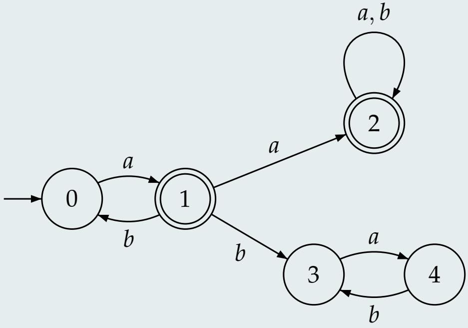
Anticipatory

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

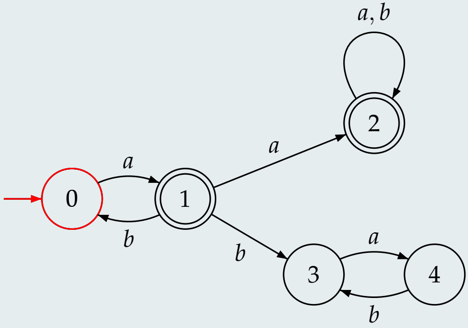
$$\begin{array}{lcl} \epsilon & \models & XXXfalse \\ a & \models & XXfalse \\ aa & \models & Xfalse \\ aaa & \models & false \end{array}$$

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

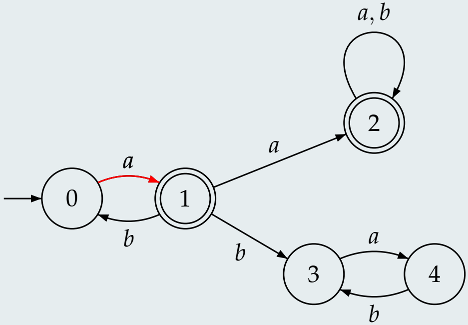
Büchi automata (BA)



Büchi automata (BA)

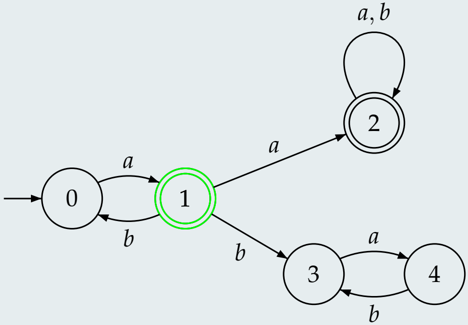


Büchi automata (BA)



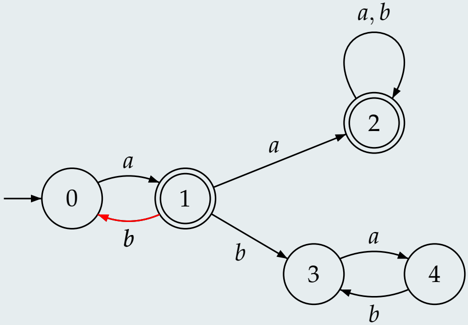
a

Büchi automata (BA)



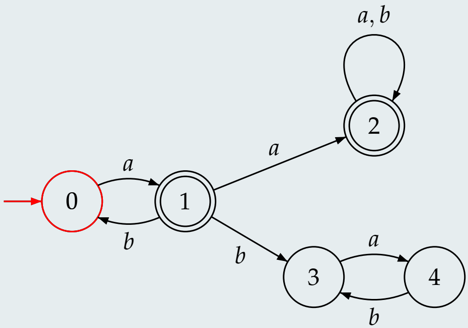
a

Büchi automata (BA)



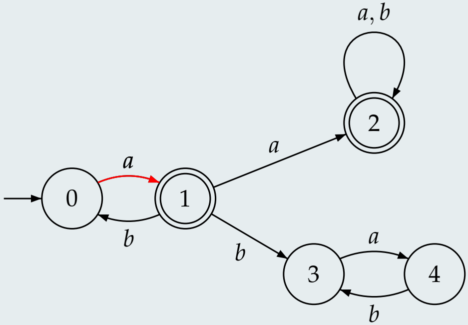
$a\ b$

Büchi automata (BA)



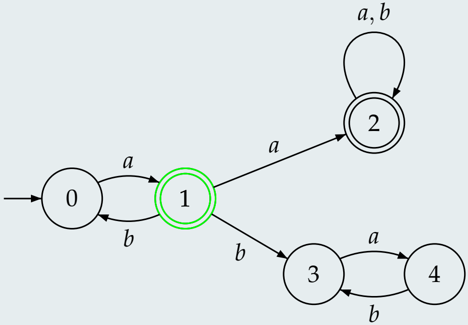
$a\ b$

Büchi automata (BA)



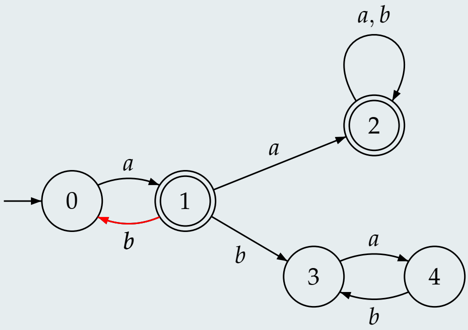
$a\ b\ a$

Büchi automata (BA)



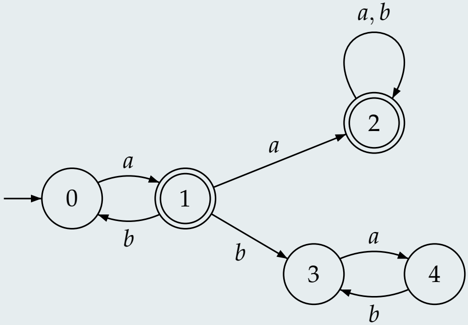
$a\ b\ a$

Büchi automata (BA)



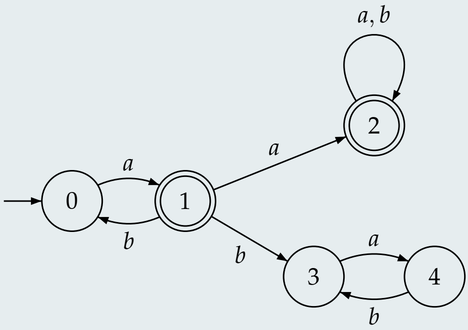
$a\ b\ a\ b$

Büchi automata (BA)



$a\ b\ a\ b\ \ldots$

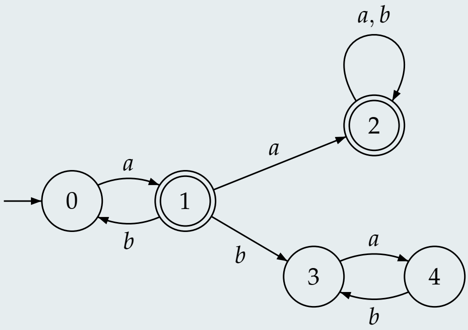
Büchi automata (BA)



$a\ b\ a\ b\ \ldots$

$(ab)^\omega \in \mathcal{L}(\mathcal{A})$

Büchi automata (BA)



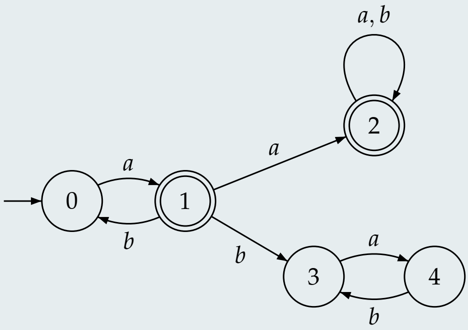
$a\ b\ a\ b\ \ldots$

$(ab)^\omega \in \mathcal{L}(\mathcal{A})$

$(ab)^* aa\{a,b\}^\omega \subseteq \mathcal{L}(\mathcal{A})$

Büchi automata (BA)

Emptiness test:



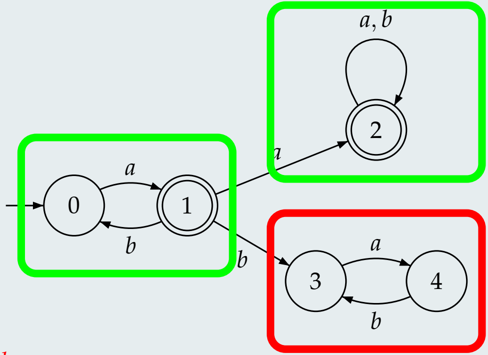
a b a b . . .

$(ab)^\omega \in \mathcal{L}(\mathcal{A})$

$(ab)^* aa\{a,b\}^\omega \subseteq \mathcal{L}(\mathcal{A})$

Büchi automata (BA)

Emptiness test: SCCC, Tarjan



$a\ b\ a\ b\ \ldots$

$(ab)^\omega \in \mathcal{L}(\mathcal{A})$

$(ab)^* aa\{a, b\}^\omega \subseteq \mathcal{L}(\mathcal{A})$

LTL to BA

[Vardi & Wolper '86]

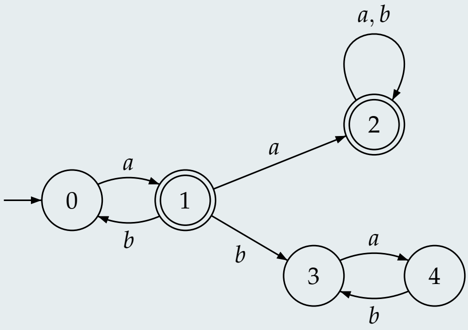
- Translation of an LTL formula φ into Büchi automata $\mathcal{A}_\varphi$ with

$$\mathcal{L}(\mathcal{A}_\varphi) = \mathcal{L}(\varphi)$$

- Complexity: Exponential in the length of φ

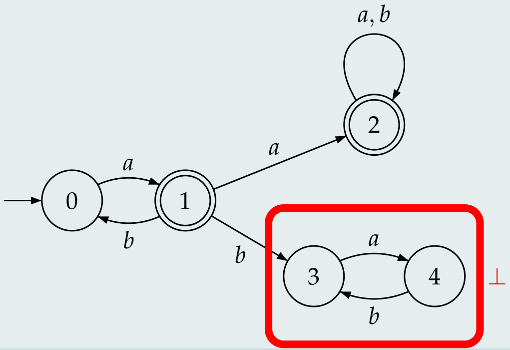
Monitor construction – Idea I

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



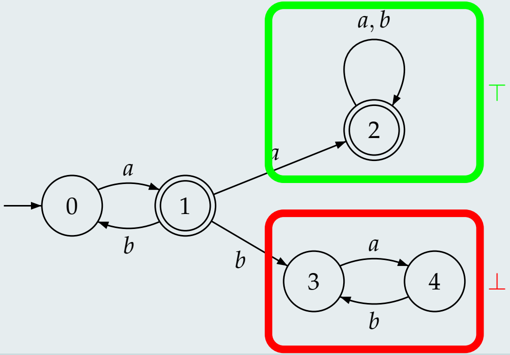
Monitor construction – Idea I

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



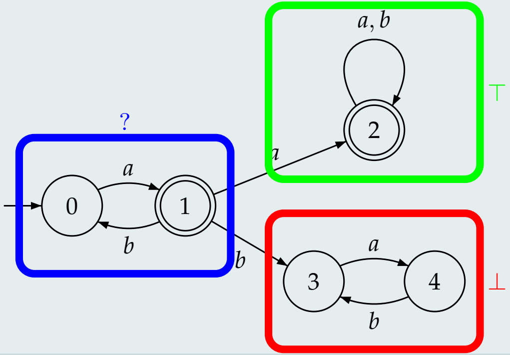
Monitor construction – Idea I

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

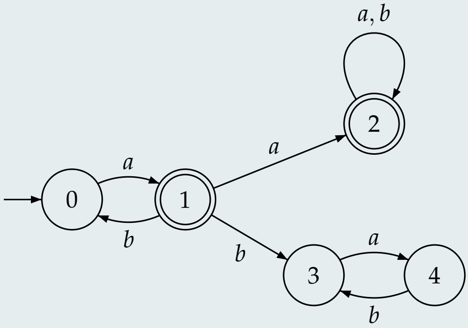


Monitor construction – Idea I

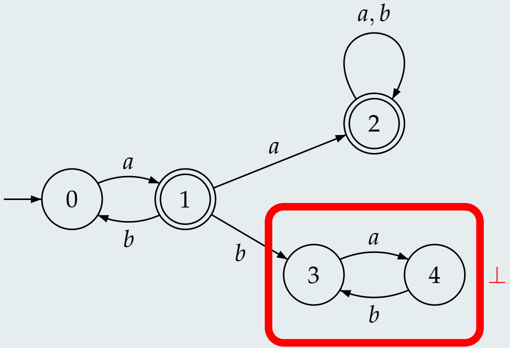
$$[u \models \varphi] = \begin{cases} \top & \text{if } \forall \sigma \in \Sigma^\omega : u\sigma \models \varphi \\ \bot & \text{if } \forall \sigma \in \Sigma^\omega : u\sigma \not\models \varphi \\ ? & \text{else} \end{cases}$$



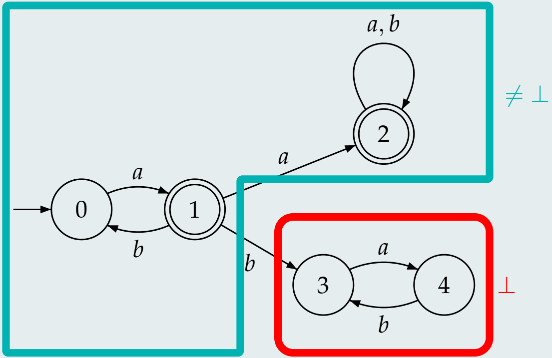
monitor construction – Idea II



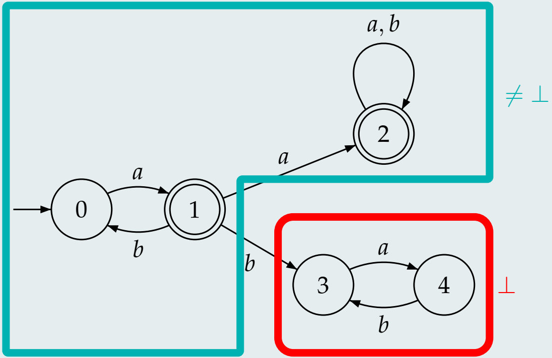
monitor construction – Idea II



monitor construction – Idea II



monitor construction – Idea II



NFA

$\mathcal{F}_\varphi : Q_\varphi \to \{\top, \bot\}$ Emptiness per state

The complete construction

The construction

$$\varphi \longrightarrow \mathrm{BA}^{\varphi} \longrightarrow \mathcal{F}^{\varphi} \longrightarrow \mathrm{NFA}^{\varphi}$$

Lemma

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

The complete construction

The construction

$$\varphi \longrightarrow \mathrm{BA}^{\varphi} \longrightarrow \mathcal{F}^{\varphi} \longrightarrow \mathrm{NFA}^{\varphi}$$

$$\neg\varphi$$

Lemma

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

The complete construction

The construction

$$\varphi \longrightarrow \mathrm{BA}^{\varphi} \longrightarrow \mathcal{F}^{\varphi} \longrightarrow \mathrm{NFA}^{\varphi}$$

$$\neg\varphi \longrightarrow \mathrm{BA}^{\neg\varphi} \longrightarrow \mathcal{F}^{\neg\varphi} \longrightarrow \mathrm{NFA}^{\neg\varphi}$$

Lemma

$$[u \models \varphi] = \begin{cases} \top & \text{if } u \notin \mathcal{L}(\mathrm{NFA}^{\neg\varphi}) \\ \bot & \text{if } u \notin \mathcal{L}(\mathrm{NFA}^{\varphi}) \\ ? & \text{else} \end{cases}$$

The complete construction

The construction

$$\varphi \longrightarrow \varphi \longrightarrow \mathrm{BA}^{\varphi} \longrightarrow \mathcal{F}^{\varphi} \longrightarrow \mathrm{NFA}^{\varphi}$$

$$\varphi \longrightarrow \neg\varphi \longrightarrow \mathrm{BA}^{\neg\varphi} \longrightarrow \mathcal{F}^{\neg\varphi} \longrightarrow \mathrm{NFA}^{\neg\varphi}$$

The complete construction

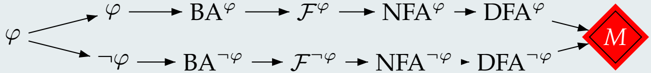
The construction

$$\varphi \longrightarrow \varphi \longrightarrow \mathrm{BA}^{\varphi} \longrightarrow \mathcal{F}^{\varphi} \longrightarrow \mathrm{NFA}^{\varphi} \longrightarrow \mathrm{DFA}^{\varphi}$$

$$\varphi \longrightarrow \neg\varphi \longrightarrow \mathrm{BA}^{\neg\varphi} \longrightarrow \mathcal{F}^{\neg\varphi} \longrightarrow \mathrm{NFA}^{\neg\varphi} \longrightarrow \mathrm{DFA}^{\neg\varphi}$$

The complete construction

The construction



Static initialisation order fiasco

$\neg spawn \mathcal{U} init$ $\qquad$ $\neg(\neg spawn \mathcal{U} init)$

Static initialisation order fiasco

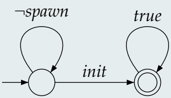
$\neg spawnUinit$ $\qquad$ $\neg(\neg spawnUinit)$

$\downarrow$ $\qquad$ $\downarrow$

Static initialisation order fiasco

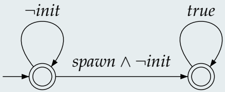
$\neg spawn \mathcal{U} init$

$\downarrow$



$\neg(\neg spawn \mathcal{U} init)$

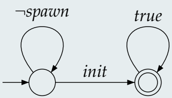
$\downarrow$



Static initialisation order fiasco

$\neg spawn \mathcal{U} init$

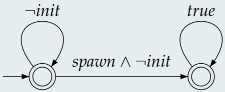
$\downarrow$



$\downarrow$

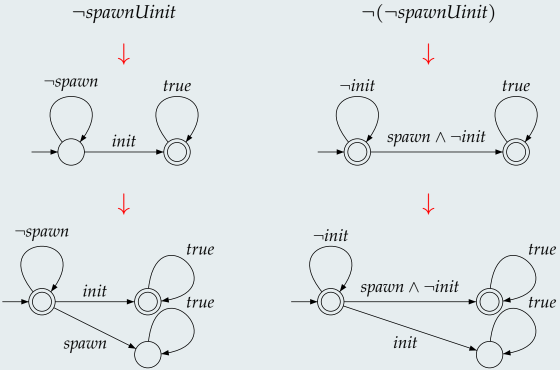
$\neg(\neg spawn \mathcal{U} init)$

$\downarrow$

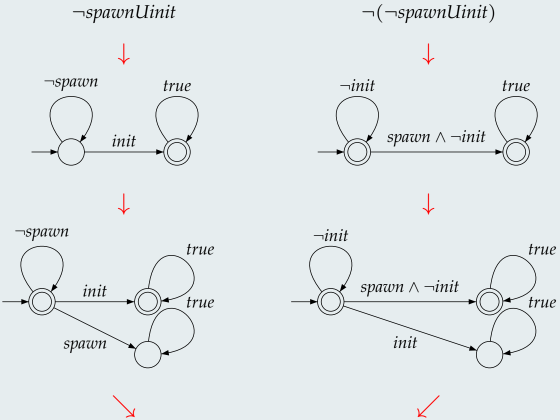


$\downarrow$

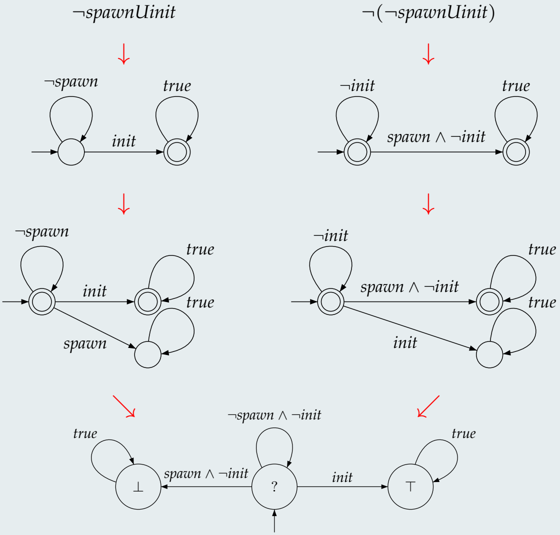
Static initialisation order fiasco



Static initialisation order fiasco

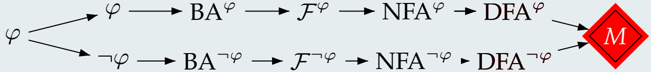


Static initialisation order fiasco



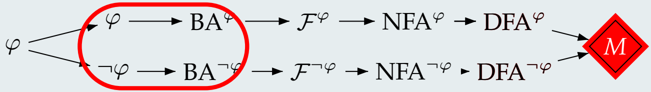
Complexity

The construction



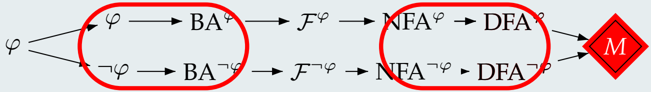
Complexity

The construction



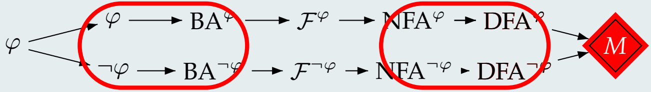
Complexity

The construction



Complexity

The construction

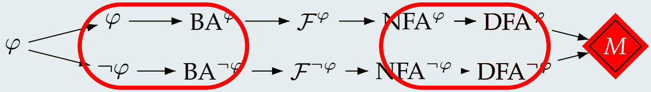


Complexity

$$|M| \leq 2^{2^{|\varphi|}}$$

Complexity

The construction



Complexity

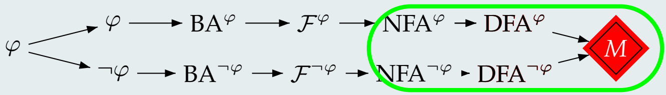
$$|M| \leq 2^{2^{|\varphi|}}$$

Optimal result!

FSM can be minimised (Myhill-Nerode)

On-the-fly Construction

The construction



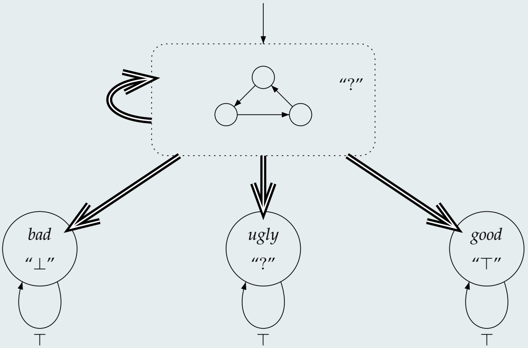
Outline

Monitorability

When does anticipation help?

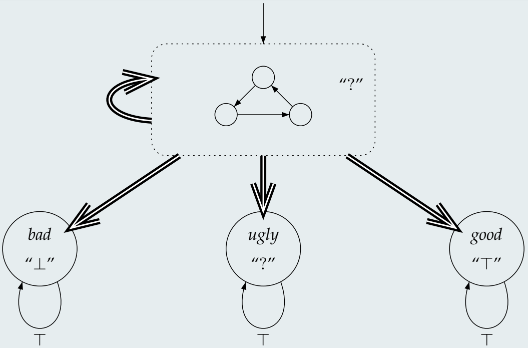
Monitors revisited

Structure of Monitors



Monitors revisited

Structure of Monitors

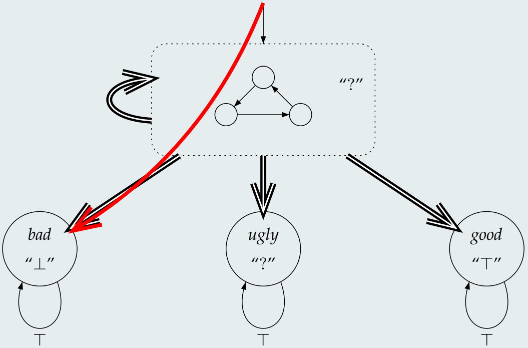


Classification of Prefixes of Words

- Bad prefixes [Kupferman & Vardi'01]

Monitors revisited

Structure of Monitors

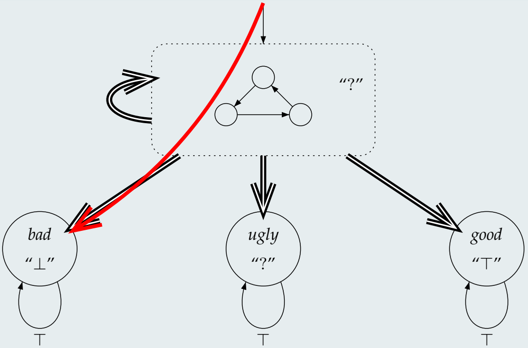


Classification of Prefixes of Words

- Bad prefixes [Kupferman & Vardi'01]

Monitors revisited

Structure of Monitors

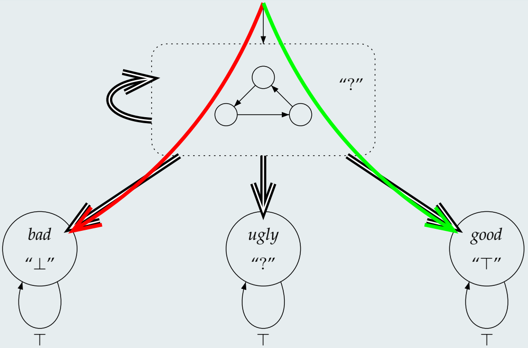


Classification of Prefixes of Words

- Bad prefixes [Kupferman & Vardi'01]
- Good prefixes [Kupferman & Vardi'01]

Monitors revisited

Structure of Monitors

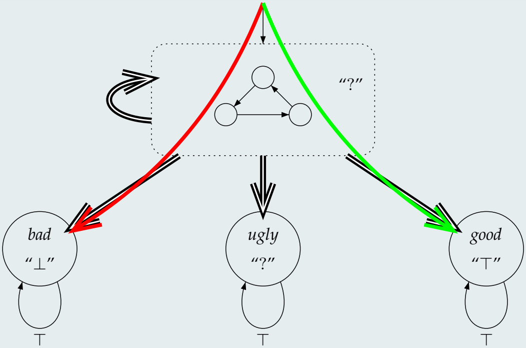


Classification of Prefixes of Words

- Bad prefixes [Kupferman & Vardi'01]
- Good prefixes [Kupferman & Vardi'01]

Monitors revisited

Structure of Monitors

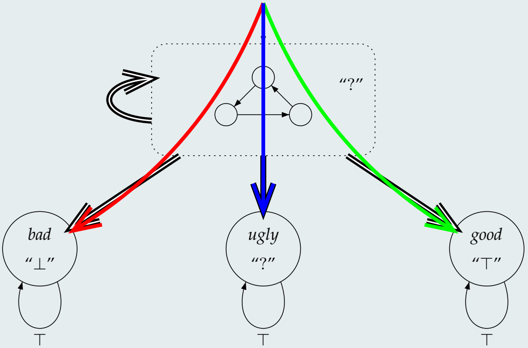


Classification of Prefixes of Words

- Bad prefixes [Kupferman & Vardi'01]
- Good prefixes [Kupferman & Vardi'01]
- Ugly prefixes

Monitors revisited

Structure of Monitors



Classification of Prefixes of Words

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

Monitorable

Non-Monitorable [Pnueli & Zaks'07]

φ is non-monitorable after u, if u cannot be extended to a bad oder good prefix.

Monitorable

φ is monitorable if there is no such u.

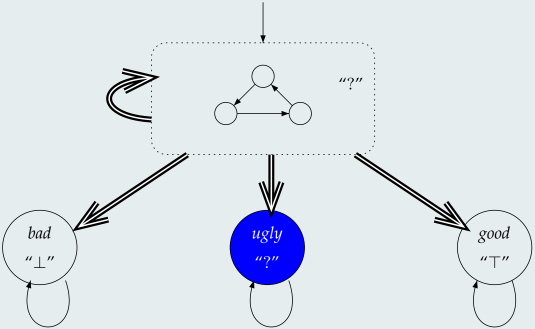
Monitorable

Non-Monitorable [Pnueli & Zaks'07]

φ is non-monitorable after u, if u cannot be extended to a bad oder good prefix.

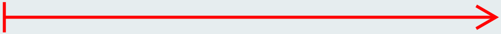
Monitorable

φ is monitorable if there is no such u.



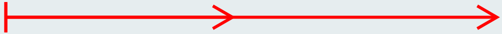
Monitorable Properties

Safety Properties



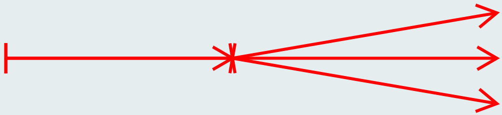
Monitorable Properties

Safety Properties



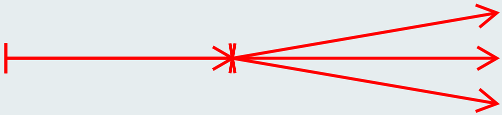
Monitorable Properties

Safety Properties



Monitorable Properties

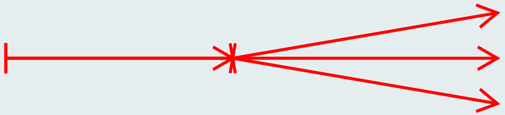
Safety Properties



Co-Safety Properties

Monitorable Properties

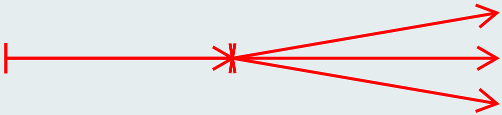
Safety Properties



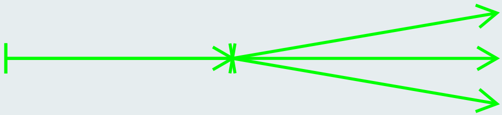
Co-Safety Properties

Monitorable Properties

Safety Properties

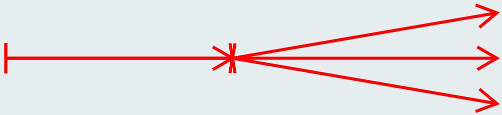


Co-Safety Properties

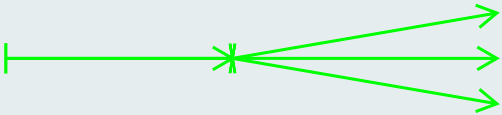


Monitorable Properties

Safety Properties



Co-Safety Properties



Note

Safety and Co-Safety Properties are monitorable

Safety- and Co-Safety-Properties

Theorem

The class of monitorable properties

- comprises safety- and co-safety properties, but
- is strictly larger than their union.

Proof

Consider $((p \vee q)Ur) \vee Gp$

Outline

RV-LTL

Basic idea

- Use LTL_3 for $\top$ and $\bot$, use FLTL_4 or 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

Fusing model checking and runtime verification

LTL with a predictive semantics

Recall anticipatory LTL semantics

The truth value of a LTL_3 formula φ 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 $\hat{\mathcal{P}}$ 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 φ wrt. $\hat{\mathcal{P}}$, denoted by $[u \models_{\hat{\mathcal{P}}} \varphi]$, is an element of $\mathbb{B}_3 \uplus \{¿\}$ and defined as follows:

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

Assuming program is known, applied to the empty word

Empty word ϵ

$$[\epsilon \models \varphi]_{\mathcal{P}} = \top$$

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

iff $\mathcal{L}(\mathcal{P}) \models \varphi$

RV more difficult than MC?

Then runtime verification implicitly answers model checking

Abstraction

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 $\mathcal{P}$ be a program and let $\hat{\mathcal{P}}$ be an over-approximation of $\mathcal{P}$. Let $u \in \Sigma^*$ denote a finite trace. The *truth value* of u and an LTL_3 formula φ wrt. $\hat{\mathcal{P}}$, denoted by $[u \models_{\hat{\mathcal{P}}} \varphi]$, is an element of $\mathbb{B}_3$ and defined as follows:

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

We write $\text{LTL}_{\mathcal{P}}$ whenever we consider LTL formulas with a predictive semantics.

Properties of Predictive Semantics

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

- Model checking is more precise than RV with the predictive semantics:

$$\mathcal{P} \models \varphi \text{ implies } [u \models_{\hat{\mathcal{P}}} \varphi] \in \{\top, ?\}$$

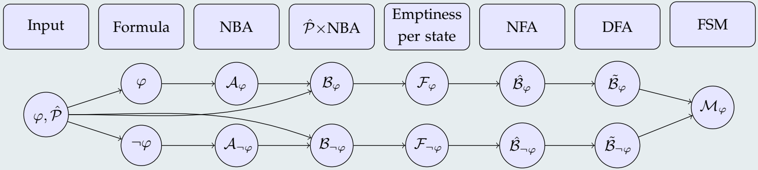
- RV has no false negatives: $[u \models_{\hat{\mathcal{P}}} \varphi] = \bot$ implies $\mathcal{P} \not\models \varphi$
- The predictive semantics of an LTL formula is more precise than LTL_3:

$$\begin{aligned} [u \models \varphi] = \top \quad &\text{implies} \quad [u \models_{\hat{\mathcal{P}}} \varphi] = \top \\ [u \models \varphi] = \bot \quad &\text{implies} \quad [u \models_{\hat{\mathcal{P}}} \varphi] = \bot \end{aligned}$$

The reverse directions are in general not true.

Monitor generation

The procedure for getting $[u \models_{\hat{\mathcal{P}}} \varphi]$ for a given φ and over-approximation $\hat{\mathcal{P}}$



Outline

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

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

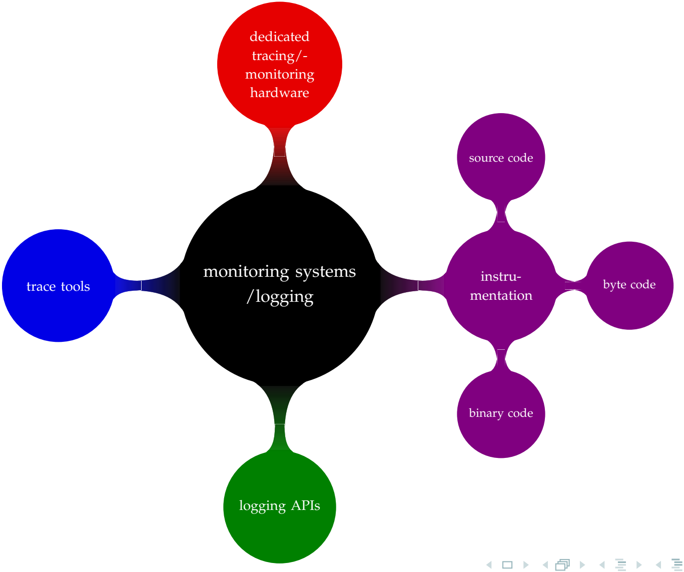
- LOLA
- Eagle (etc.)

Further dimensions

- real-time
- concurrency
- distribution

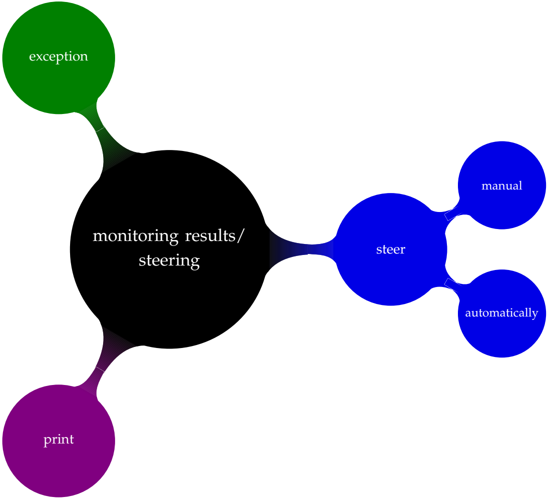
Presentation outline

Monitoring Systems/Logging: Overview



Presentation outline

Monitoring Systems/Logging: Overview



React!

Runtime Verification

Observe—do not react

Realising dynamic systems

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

React!

Runtime Verification

Observe—do not react

Realising dynamic systems

- self-healing systems
- adaptive systems, self-organising systems
- ...
- use monitors for observation—then react

jMOP [Rosu et al.]

Java Implementation

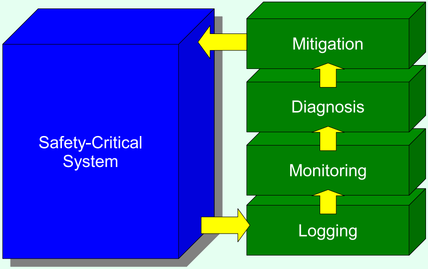
```
class Resource {
/*@
     scope = class
     logic = PTLTL
     {
      Event authenticate: end(exec(*
authenticate()));
      Event use: begin(exec(* access()));
      Formula : use -> <*> authenticate
     }
     violation Handler {
       @this.authenticate();
     }
@*/
void authenticate() {...}
void access() {...}
...
}
```

Where → How → what → What if →

Runtime Reflection [Bauer, L., Schallhart@ASWEC'06]

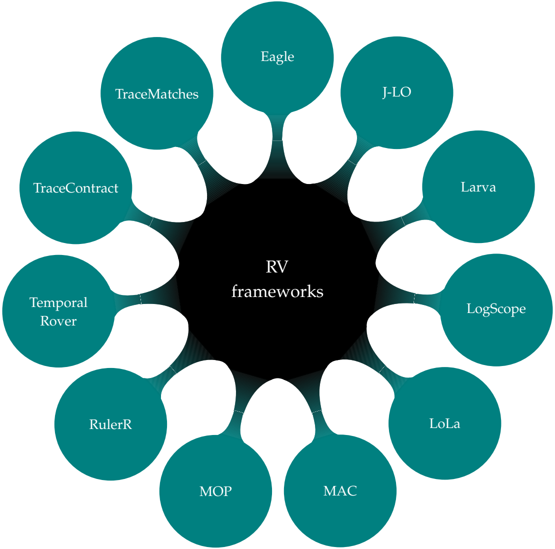
Monitor-based Runtime Reflection

Software Architecture Pattern



Presentation outline

Monitoring Systems/Logging: Overview



Presentation outline

Outline

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:

Example

```
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

```
@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

Main Ideas

- seperate 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

Events and Propositions

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
- $G(\texttt{modify} \rightarrow \neg\texttt{disconnect}\, U\, \texttt{committed})$

Example (Specifying Monitors)

```
private static Monitor commitBeforeDisconnect = new FLTL4Monitor(
  Always(implies(
      modify,
      Until(not(disconnect), committed)
    )
  ));
```

Testcase

Example

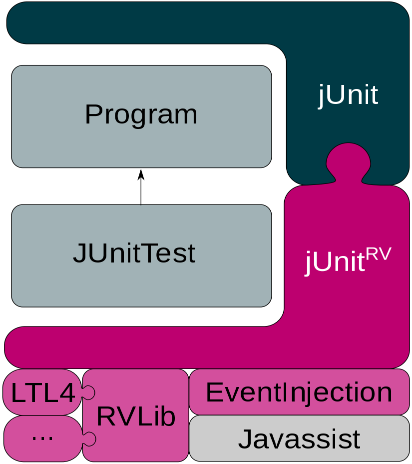
```
@Test
@Monitors({"commitBeforeDisconnect"})
public void test1() {
  DataService service = new MyDataService("http://myserver.net");
  MyDataClient client = new MyDataClient(service);

  client.authenticate("daniel");
  client.addPatient("Mr. Smith");
  client.switchToUser("ruth");
  client.getPatientFile("miller-2143-1");
  client.setPhone("miller-2143-1", "012345678");
  client.exit();
}
```

The Complete Picture

```
@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() {
        ...
    }
}
```

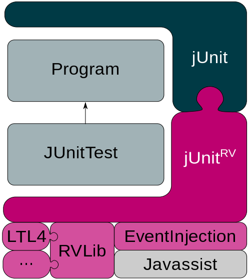
Outline

Architecture



Runners and Classloaders

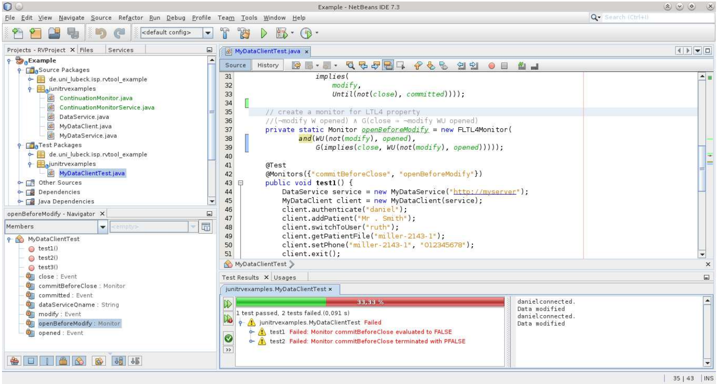
- jUnit uses test runners to execute tests
- jUnit provides a default implementation
- jUnit[RV] provides `RVRunner` extending the default implementation
- jUnit[RV] provides a custom `Classloader`
- Class loading by program under scrutiny is intercepted
- Bytecode is manipulated to intercept events



Features

- jUnit^{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

jUnit^{RV} Running in Netbeans



jUnitRV – Summary

- Unit testing and runtime verification are combined
- jUnit is extended by temporal assertions
- Testing temporal properties is less cumbersome
- jUnit^{RV} integrates easily in existing projects and environments

Presentation outline

Conclusion

Summary

- RV needs similar temporal logics as model checking, but adaptions for
  - finite runs
  - impartiality
  - anticipation
  - prediction
- Application jUnit^{RV}

That's it!

Thanks! - Questions?