Security protocols, properties, and their monitoring

Andreas Bauer

Computer Sciences Laboratory,
The Australian National University

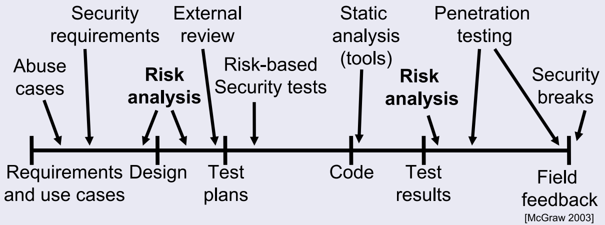
October 22, 2008

Based on work undertaken with Jan Jürjens, Martin Leucker, and Christian Schallhart.

Outline

Software and systems verification

Secure systems life-cycle

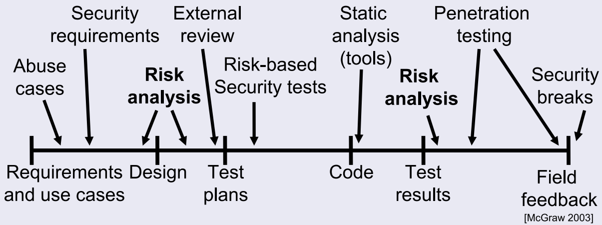


Some observations

- Static analysis (and static verification) operate on abstractions of the real-world system (code, state-models, etc.)
- Penetration testing works on actual system, but is not complete

Software and systems verification

Secure systems life-cycle

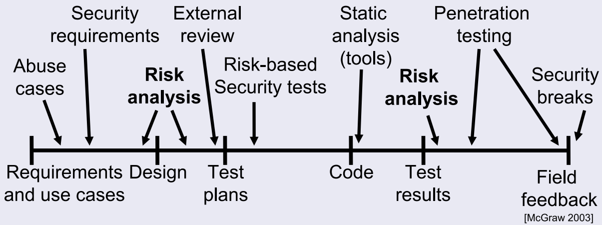


Some observations

- Static analysis (and static verification) operate on abstractions of the real-world system (code, state-models, etc.)
- Penetration testing works on actual system, but is not complete

Software and systems verification

Secure systems life-cycle

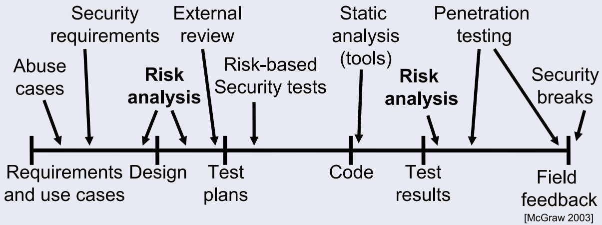


Some observations

- Static analysis (and static verification) operate on abstractions of the real-world system (code, state-models, etc.)
- Penetration testing works on actual system, but is not complete

Software and systems verification

Secure systems life-cycle

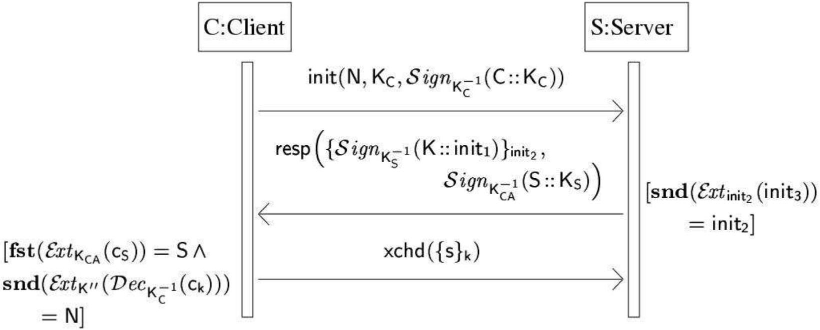


Some observations

- Static analysis (and static verification) operate on abstractions of the real-world system (code, state-models, etc.)
- Penetration testing works on actual system, but is not complete

Example: (semi-automatic) static verification

- System model, e.g., UML message sequence chart (MSC) of a protocol



Example: (semi-automatic) static verification

- System model, e.g., UML message sequence chart (MSC) of a protocol
- Predicate knows(E) meaning that adversary may get to know E during the execution of the system
- E.g. secrecy requirement: For any secret s, check whether can derive knows(s)
- Automatically generate behavioural model of protocol (e.g. from UMLsec)
- Formalise security property, e.g.:
- Use theorem prover to check model against property

Example: (semi-automatic) static verification

- System model, e.g., UML message sequence chart (MSC) of a protocol
- Predicate knows(E) meaning that adversary may get to know E during the execution of the system
- E.g. secrecy requirement: For any secret s, check whether can derive knows(s)
- Automatically generate behavioural model of protocol (e.g. from UMLsec)
- Formalise security property, e.g.:
- Use theorem prover to check model against property

Example: (semi-automatic) static verification

- System model, e.g., UML message sequence chart (MSC) of a protocol
- Predicate knows(E) meaning that adversary may get to know E during the execution of the system
- E.g. secrecy requirement: For any secret s, check whether can derive knows(s)
- Automatically generate behavioural model of protocol (e.g. from UMLsec)
- Formalise security property, e.g.:
- Use theorem prover to check model against property

Example: (semi-automatic) static verification

- System model, e.g., UML message sequence chart (MSC) of a protocol
- Predicate knows(E) meaning that adversary may get to know E during the execution of the system
- E.g. secrecy requirement: For any secret s, check whether can derive knows(s)
- Automatically generate behavioural model of protocol (e.g. from UMLsec)
- Formalise security property, e.g.:

$$
\begin{aligned}
& knows(N) \wedge knows(K_C) \wedge knows(Sign_{K_C^{-1}}(C::K_C)) \\
& \quad \wedge \forall init_1, init_2, init_3. [knows(init_1) \wedge knows(init_2) \wedge \\
& \qquad knows(init_3) \wedge snd(Ext_{init_2}(init_3)) = init_2 \\
& \qquad) \; knows(\{Sign_{K_S^{-1}}(\ldots)\}_{..}) \wedge [knows(Sign \ldots)] \\
& \quad \wedge \forall resp_1, resp_2. \; [\ldots) \ldots]]
\end{aligned}
$$

- Use theorem prover to check model against property

Example: (semi-automatic) static verification

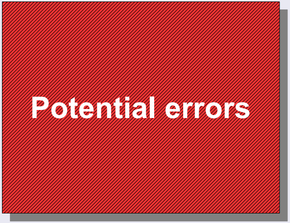
- System model, e.g., UML message sequence chart (MSC) of a protocol
- Predicate knows(E) meaning that adversary may get to know E during the execution of the system
- E.g. secrecy requirement: For any secret s, check whether can derive knows(s)
- Automatically generate behavioural model of protocol (e.g. from UMLsec)
- Formalise security property, e.g.:

$$\begin{aligned}
& knows(N) \wedge knows(K_C) \wedge knows(Sign_{K_C^{-1}}(C::K_C)) \\
& \quad \wedge \forall init_1, init_2, init_3. [knows(init_1) \wedge knows(init_2) \wedge \\
& \qquad knows(init_3) \wedge snd(Ext_{init_2}(init_3)) = init_2 \\
& \qquad) \; knows(\{Sign_{K_S^{-1}}(\ldots)\}_{..}) \wedge [knows(Sign \ldots)] \\
& \quad \wedge \forall resp_1, resp_2. \; [\ldots) \ldots]]
\end{aligned}$$

- Use theorem prover to check model against property

Monitoring/runtime verification

Mind the gap!



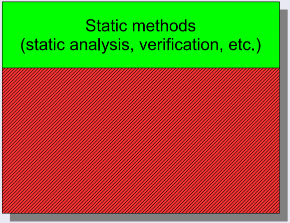
- "Red area" typically not even finite, because systems are often infinite state systems (interaction with environment, real-time, etc.)
- Often impossible to give a 100% guarantee for safety or security

Monitoring/runtime verification "sits in the gap"

- Dynamic verification, operates on actual system
- Checks actual system behaviour against correctness property
- Ensures that statically verified properties hold at runtime

Monitoring/runtime verification

Mind the gap!



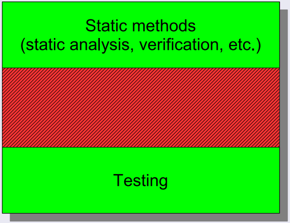
- "Red area" typically not even finite, because systems are often infinite state systems (interaction with environment, real-time, etc.)
- Often impossible to give a 100% guarantee for safety or security

Monitoring/runtime verification "sits in the gap"

- Dynamic verification, operates on actual system
- Checks actual system behaviour against correctness property
- Ensures that statically verified properties hold at runtime

Monitoring/runtime verification

Mind the gap!



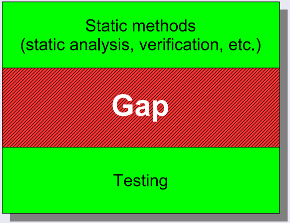
- "Red area" typically not even finite, because systems are often infinite state systems (interaction with environment, real-time, etc.)
- Often impossible to give a 100% guarantee for safety or security

Monitoring/runtime verification "sits in the gap"

- Dynamic verification, operates on actual system
- Checks actual system behaviour against correctness property
- Ensures that statically verified properties hold at runtime

Monitoring/runtime verification

Mind the gap!



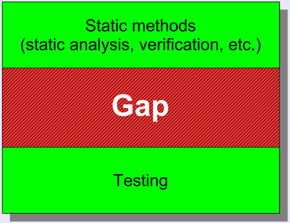
- "Red area" typically not even finite, because systems are often infinite state systems (interaction with environment, real-time, etc.)
- Often impossible to give a 100% guarantee for safety or security

Monitoring/runtime verification "sits in the gap"

- Dynamic verification, operates on actual system
- Checks actual system behaviour against correctness property
- Ensures that statically verified properties hold at runtime

Monitoring/runtime verification

Mind the gap!



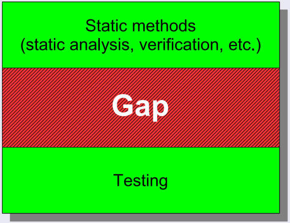
- "Red area" typically not even finite, because systems are often infinite state systems (interaction with environment, real-time, etc.)
- Often impossible to give a 100% guarantee for safety or security

Monitoring/runtime verification "sits in the gap"

- Dynamic verification, operates on actual system
- Checks actual system behaviour against correctness property
- Ensures that statically verified properties hold at runtime

Monitoring/runtime verification

Mind the gap!



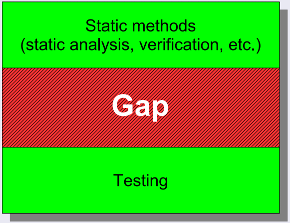
- "Red area" typically not even finite, because systems are often infinite state systems (interaction with environment, real-time, etc.)
- Often impossible to give a 100% guarantee for safety or security

Monitoring/runtime verification "sits in the gap"

- Dynamic verification, operates on actual system
- Checks actual system behaviour against correctness property
- Ensures that statically verified properties hold at runtime

Monitoring/runtime verification

Mind the gap!



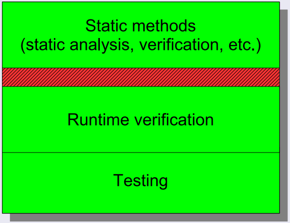
- "Red area" typically not even finite, because systems are often infinite state systems (interaction with environment, real-time, etc.)
- Often impossible to give a 100% guarantee for safety or security

Monitoring/runtime verification "sits in the gap"

- Dynamic verification, operates on actual system
- Checks actual system behaviour against correctness property
- Ensures that statically verified properties hold at runtime

Monitoring/runtime verification

Mind the gap!



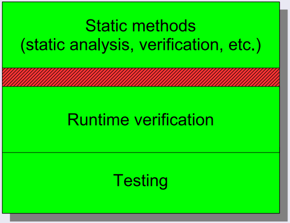
- "Red area" typically not even finite, because systems are often infinite state systems (interaction with environment, real-time, etc.)
- Often impossible to give a 100% guarantee for safety or security

Monitoring/runtime verification "sits in the gap"

- Dynamic verification, operates on actual system
- Checks actual system behaviour against correctness property
- Ensures that statically verified properties hold at runtime

Monitoring/runtime verification

Mind the gap!



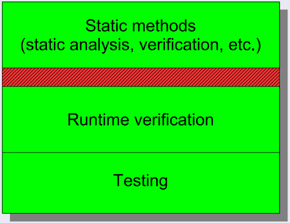
- "Red area" typically not even finite, because systems are often infinite state systems (interaction with environment, real-time, etc.)
- Often impossible to give a 100% guarantee for safety or security

Monitoring/runtime verification "sits in the gap"

- Dynamic verification, operates on actual system
- Checks actual system behaviour against correctness property
- Ensures that statically verified properties hold at runtime

Monitoring/runtime verification

Mind the gap!



- "Red area" typically not even finite, because systems are often infinite state systems (interaction with environment, real-time, etc.)
- Often impossible to give a 100% guarantee for safety or security

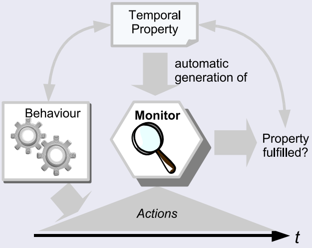
Monitoring/runtime verification "sits in the gap"

- Dynamic verification, operates on actual system
- Checks actual system behaviour against correctness property
- Ensures that statically verified properties hold at runtime

Runtime verification—how it's done

Runtime verification—how it's done

Central concept: monitoring of actions



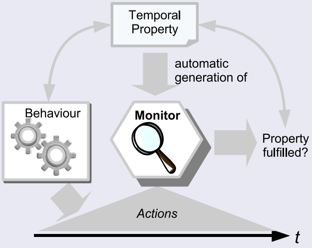
- Property, φ, specified in terms of LTL(Σ) [Pnu77], where $\Sigma = 2^{AP}$:
  - $\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \mathbf{U} \varphi \mid \mathbf{X}\varphi$, with $p \in AP$
- Interpretation of φ over linearly growing stream of actions, $u \in \Sigma^*$:
  - Monitor: $[u \models \varphi] = \top$?.

Central research questions

- Complexity of monitor generation usually irrelevant
- How to generate good monitors?
- What are suitable logics for property specification?
- And what are their properties?

Runtime verification—how it's done

Central concept: monitoring of actions



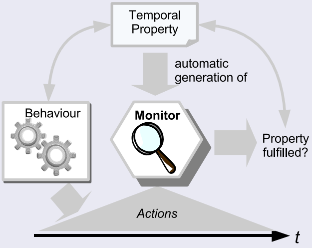
- Property, φ, specified in terms of LTL(Σ) [Pnu77], where $\Sigma = 2^{AP}$:
  - $\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \mathbf{U} \varphi \mid \mathbf{X}\varphi$, with $p \in AP$
- Interpretation of φ over linearly growing stream of actions, $u \in \Sigma^*$:
  - Monitor: $[u \models \varphi] = \top$?.

Central research questions

- Complexity of monitor generation usually irrelevant
- How to generate good monitors?
- What are suitable logics for property specification?
- And what are their properties?

Runtime verification—how it's done

Central concept: monitoring of actions



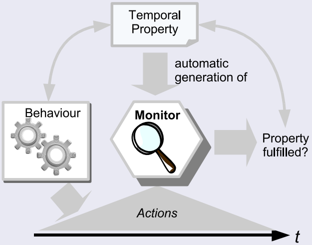
- Property, φ, specified in terms of LTL(Σ) [Pnu77], where $\Sigma = 2^{AP}$:
  - $\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \mathbf{U} \varphi \mid \mathbf{X}\varphi$, with $p \in AP$
- Interpretation of φ over linearly growing stream of actions, $u \in \Sigma^*$:
  - Monitor: $[u \models \varphi] = \top$?.

Central research questions

- Complexity of monitor generation usually irrelevant
- How to generate good monitors?
- What are suitable logics for property specification?
- And what are their properties?

Runtime verification—how it's done

Central concept: monitoring of actions



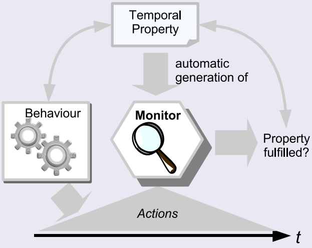
- Property, φ, specified in terms of LTL(Σ) [Pnu77], where $\Sigma = 2^{AP}$:
  - $\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \mathbf{U} \varphi \mid \mathbf{X}\varphi$, with $p \in AP$
- Interpretation of φ over linearly growing stream of actions, $u \in \Sigma^*$:
  - Monitor: $[u \models \varphi] = \top$?.

Central research questions

- Complexity of monitor generation usually irrelevant
- How to generate good monitors?
- What are suitable logics for property specification?
- And what are their properties?

Runtime verification—how it's done

Central concept: monitoring of actions



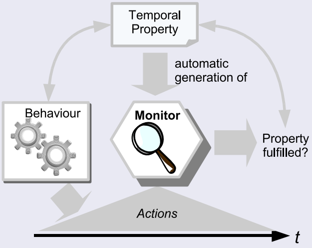
- Property, φ, specified in terms of LTL(Σ) [Pnu77], where $\Sigma = 2^{AP}$:
  - $\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \mathbf{U} \varphi \mid \mathbf{X}\varphi$, with $p \in AP$
- Interpretation of φ over linearly growing stream of actions, $u \in \Sigma^*$:
  - Monitor: $[u \models \varphi] = \top$?.

Central research questions

- Complexity of monitor generation usually irrelevant
- How to generate good monitors?
- What are suitable logics for property specification?
- And what are their properties?

Runtime verification—how it's done

Central concept: monitoring of actions



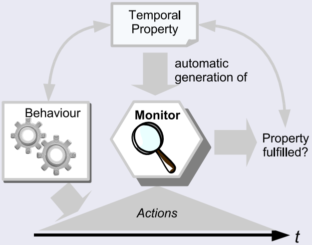
- Property, φ, specified in terms of LTL(Σ) [Pnu77], where $\Sigma = 2^{AP}$:
  - $\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \mathbf{U} \varphi \mid \mathbf{X}\varphi$, with $p \in AP$
- Interpretation of φ over linearly growing stream of actions, $u \in \Sigma^*$:
  - Monitor: $[u \models \varphi] = \top$?.

Central research questions

- Complexity of monitor generation usually irrelevant
- How to generate good monitors?
- What are suitable logics for property specification?
- And what are their properties?

Runtime verification—how it's done

Central concept: monitoring of actions



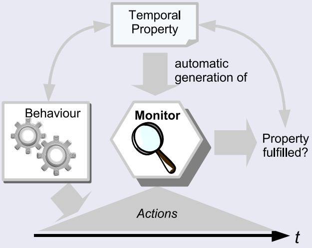
- Property, φ, specified in terms of LTL(Σ) [Pnu77], where $\Sigma = 2^{AP}$:
  - $\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \mathbf{U} \varphi \mid \mathbf{X}\varphi$, with $p \in AP$
- Interpretation of φ over linearly growing stream of actions, $u \in \Sigma^*$:
  - Monitor: $[u \models \varphi] = \top$?.

Central research questions

- Complexity of monitor generation usually irrelevant
- How to generate good monitors?
- What are suitable logics for property specification?
- And what are their properties?

Runtime verification—how it's done

Central concept: monitoring of actions



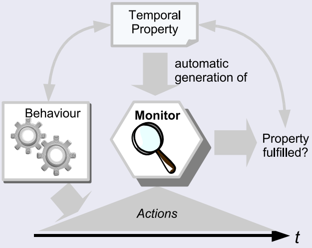
- Property, φ, specified in terms of LTL(Σ) [Pnu77], where $\Sigma = 2^{AP}$:
  - $\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \mathbf{U} \varphi \mid \mathbf{X}\varphi$, with $p \in AP$
- Interpretation of φ over linearly growing stream of actions, $u \in \Sigma^*$:
  - Monitor: $[u \models \varphi] = \top$?.

Central research questions

- Complexity of monitor generation usually irrelevant
- How to generate good monitors?
- What are suitable logics for property specification?
- And what are their properties?

Runtime verification—how it's done

Central concept: monitoring of actions



- Property, φ, specified in terms of LTL(Σ) [Pnu77], where $\Sigma = 2^{AP}$:
  - $\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \mathbf{U} \varphi \mid \mathbf{X}\varphi$, with $p \in AP$
- Interpretation of φ over linearly growing stream of actions, $u \in \Sigma^*$:
  - Monitor: $[u \models \varphi] = \top$?.

Central research questions

- Complexity of monitor generation usually irrelevant
- How to generate good monitors?
- What are suitable logics for property specification?
- And what are their properties?

What can be specified?

Let $\varphi \in \text{LTL}(\Sigma)$ be an LTL formula, and $i \in \mathbb{N}$ denote a position.

Formal LTL semantics

The *semantics of LTL* formulae is defined inductively over infinite strings $w \in \Sigma^\omega$ as follows:

$$
\begin{array}{lcl}
w, i \models true & & \\
w, i \models \neg\varphi & \Leftrightarrow & w, i \not\models \varphi \\
w, i \models p \in AP & \Leftrightarrow & p \in w(i) \\
w, i \models \varphi_1 \vee \varphi_2 & \Leftrightarrow & w, i \models \varphi_1 \vee w, i \models \varphi_2 \\
w, i \models \varphi_1 \mathbf{U} \varphi_2 & \Leftrightarrow & \exists k \geq i.\ w, k \models \varphi_2 \wedge \\
 & & \forall i \leq l < k.\ w, l \models \varphi_1 \\
w, i \models \mathbf{X}\varphi & \Leftrightarrow & w, i+1 \models \varphi
\end{array}
$$

Notation: $w \models \varphi$, if and only if $w, 0 \models \varphi$, and $w(i)$ to denote the ith element in w.

What can be specified?

Let $\varphi \in \mathrm{LTL}(\Sigma)$ be an LTL formula, and $i \in \mathbb{N}$ denote a position.

Formal LTL semantics

The *semantics of LTL* formulae is defined inductively over infinite strings $w \in \Sigma^\omega$ as follows:

$$
\begin{array}{lcl}
w, i \models \mathit{true} & & \\
w, i \models \neg\varphi & \Leftrightarrow & w, i \not\models \varphi \\
w, i \models p \in AP & \Leftrightarrow & p \in w(i) \\
w, i \models \varphi_1 \vee \varphi_2 & \Leftrightarrow & w, i \models \varphi_1 \vee w, i \models \varphi_2 \\
w, i \models \varphi_1 \mathbf{U} \varphi_2 & \Leftrightarrow & \exists k \geq i.\ w, k \models \varphi_2 \wedge \\
 & & \forall i \leq l < k.\ w, l \models \varphi_1 \\
w, i \models \mathbf{X}\varphi & \Leftrightarrow & w, i+1 \models \varphi
\end{array}
$$

Notation: $w \models \varphi$, if and only if $w, 0 \models \varphi$, and $w(i)$ to denote the ith element in w.

What can be specified?

Let $\varphi \in \text{LTL}(\Sigma)$ be an LTL formula, and $i \in \mathbb{N}$ denote a position.

Formal LTL semantics

The *semantics of LTL* formulae is defined inductively over infinite strings $w \in \Sigma^\omega$ as follows:

$$
\begin{array}{lcl}
w, i \models true & & \\
w, i \models \neg\varphi & \Leftrightarrow & w, i \not\models \varphi \\
w, i \models p \in AP & \Leftrightarrow & p \in w(i) \\
w, i \models \varphi_1 \vee \varphi_2 & \Leftrightarrow & w, i \models \varphi_1 \vee w, i \models \varphi_2 \\
w, i \models \varphi_1 \mathbf{U} \varphi_2 & \Leftrightarrow & \exists k \geq i.\ w, k \models \varphi_2 \wedge \\
& & \forall i \leq l < k.\ w, l \models \varphi_1 \\
w, i \models \mathbf{X}\varphi & \Leftrightarrow & w, i+1 \models \varphi
\end{array}
$$

Notation: $w \models \varphi$, if and only if $w, 0 \models \varphi$, and $w(i)$ to denote the ith element in w.

What can be specified? (intuitive semantics)

"All interesting properties about a system can be expressed using safety and liveness properties." – L. Lamport, 1977.

Safety properties

- If $L \subseteq \Sigma^\omega$ is a safety language, then all $w \notin L$ have a finite bad prefix.
- Consider $\mathbf{G}\varphi$:
  - $\varphi := p$ ("always p"), then $\mathbf{G}\varphi$ is safety
  - $\varphi := \mathbf{F}p$ ("eventually p"), then $\mathbf{G}\varphi$ is not safety – *Why?*

Liveness properties

- If $L \subseteq \Sigma^\omega$ is a liveness language, then for all $u \in \Sigma^*$ there exists a $w \in \Sigma^\omega$, such that $uw \in L$.

What can be specified? (intuitive semantics)

"All interesting properties about a system can be expressed using safety and liveness properties." – L. Lamport, 1977.

Safety properties

- If $L \subseteq \Sigma^\omega$ is a safety language, then all $w \notin L$ have a finite bad prefix.
- Consider $\mathbf{G}\varphi$:
  - $\varphi := p$ ("always p"), then $\mathbf{G}\varphi$ is safety
  - $\varphi := \mathbf{F}p$ ("eventually p"), then $\mathbf{G}\varphi$ is not safety – *Why?*

Liveness properties

- If $L \subseteq \Sigma^\omega$ is a liveness language, then for all $u \in \Sigma^*$ there exists a $w \in \Sigma^\omega$, such that $uw \in L$.

What can be specified? (intuitive semantics)

"All interesting properties about a system can be expressed using safety and liveness properties." – L. Lamport, 1977.

Safety properties

- If $L \subseteq \Sigma^\omega$ is a safety language, then all $w \notin L$ have a finite bad prefix.
- Consider $\mathbf{G}\varphi$:
  - $\varphi := p$ ("always p"), then $\mathbf{G}\varphi$ is safety
  - $\varphi := \mathbf{F}p$ ("eventually p"), then $\mathbf{G}\varphi$ is not safety – *Why?*

Liveness properties

- If $L \subseteq \Sigma^\omega$ is a liveness language, then for all $u \in \Sigma^*$ there exists a $w \in \Sigma^\omega$, such that $uw \in L$.

What can be specified? (intuitive semantics)

"All interesting properties about a system can be expressed using safety and liveness properties." – L. Lamport, 1977.

Safety properties

- If $L \subseteq \Sigma^\omega$ is a safety language, then all $w \notin L$ have a finite bad prefix.
- Consider $\mathbf{G}\varphi$:
  - $\varphi := p$ ("always p"), then $\mathbf{G}\varphi$ is safety
  - $\varphi := \mathbf{F}p$ ("eventually p"), then $\mathbf{G}\varphi$ is not safety – *Why?*

Liveness properties

- If $L \subseteq \Sigma^\omega$ is a liveness language, then for all $u \in \Sigma^*$ there exists a $w \in \Sigma^\omega$, such that $uw \in L$.

What can be specified? (intuitive semantics)

"All interesting properties about a system can be expressed using safety and liveness properties." – L. Lamport, 1977.

Safety properties

- If $L \subseteq \Sigma^\omega$ is a safety language, then all $w \notin L$ have a finite bad prefix.
- Consider $\mathbf{G}\varphi$:
  - $\varphi := p$ ("always p"), then $\mathbf{G}\varphi$ is safety
  - $\varphi := \mathbf{F}p$ ("eventually p"), then $\mathbf{G}\varphi$ is not safety – *Why?*

Liveness properties

- If $L \subseteq \Sigma^\omega$ is a liveness language, then for all $u \in \Sigma^*$ there exists a $w \in \Sigma^\omega$, such that $uw \in L$.

What can be specified? (intuitive semantics)

"All interesting properties about a system can be expressed using safety and liveness properties." – L. Lamport, 1977.

Safety properties

- If $L \subseteq \Sigma^\omega$ is a safety language, then all $w \notin L$ have a finite bad prefix.
- Consider $\mathbf{G}\varphi$:
  - $\varphi := p$ ("always p"), then $\mathbf{G}\varphi$ is safety
  - $\varphi := \mathbf{F}p$ ("eventually p"), then $\mathbf{G}\varphi$ is not safety – *Why?*

Liveness properties

- If $L \subseteq \Sigma^\omega$ is a liveness language, then for all $u \in \Sigma^*$ there exists a $w \in \Sigma^\omega$, such that $uw \in L$.

What can be specified? (intuitive semantics)

"All interesting properties about a system can be expressed using safety and liveness properties." – L. Lamport, 1977.

Safety properties

- If $L \subseteq \Sigma^\omega$ is a safety language, then all $w \notin L$ have a finite bad prefix.
- Consider $\mathbf{G}\varphi$:
  - $\varphi := p$ ("always p"), then $\mathbf{G}\varphi$ is safety
  - $\varphi := \mathbf{F}p$ ("eventually p"), then $\mathbf{G}\varphi$ is not safety – *Why?*

Liveness properties

- If $L \subseteq \Sigma^\omega$ is a liveness language, then for all $u \in \Sigma^*$ there exists a $w \in \Sigma^\omega$, such that $uw \in L$.

Is that all?

Other

- Interestingly, there are properties which are neither strictly liveness nor strictly safety.

Co-safety properties

- If $L \subseteq \Sigma^\omega$ is a co-safety language, then all $w \in L$ have a finite good prefix.
- Let L be co-safety, then $\overline{L}$ is safety.
- $p\mathbf{U}q$ is co-safety
- $\mathbf{F}p$ is co-safety (but also liveness)

Is that all?

Other

- Interestingly, there are properties which are neither strictly liveness nor strictly safety.

Co-safety properties

- If $L \subseteq \Sigma^\omega$ is a co-safety language, then all $w \in L$ have a finite good prefix.
- Let L be co-safety, then $\overline{L}$ is safety.
- $p\mathbf{U}q$ is co-safety
- $\mathbf{F}p$ is co-safety (but also liveness)

Is that all?

Other

- Interestingly, there are properties which are neither strictly liveness nor strictly safety.

Co-safety properties

- If $L \subseteq \Sigma^\omega$ is a co-safety language, then all $w \in L$ have a finite good prefix.
- Let L be co-safety, then $\overline{L}$ is safety.
- $p\mathbf{U}q$ is co-safety
- $\mathbf{F}p$ is co-safety (but also liveness)

Is that all?

Other

- Interestingly, there are properties which are neither strictly liveness nor strictly safety.

Co-safety properties

- If $L \subseteq \Sigma^\omega$ is a co-safety language, then all $w \in L$ have a finite good prefix.
- Let L be co-safety, then $\overline{L}$ is safety.
- $p\mathbf{U}q$ is co-safety
- $\mathbf{F}p$ is co-safety (but also liveness)

Is that all?

Other

- Interestingly, there are properties which are neither strictly liveness nor strictly safety.

Co-safety properties

- If $L \subseteq \Sigma^\omega$ is a co-safety language, then all $w \in L$ have a finite good prefix.
- Let L be co-safety, then $\overline{L}$ is safety.
- $p\mathbf{U}q$ is co-safety
- $\mathbf{F}p$ is co-safety (but also liveness)

Is that all?

Other

- Interestingly, there are properties which are neither strictly liveness nor strictly safety.

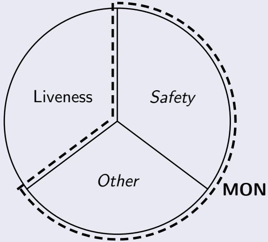
Co-safety properties

- If $L \subseteq \Sigma^\omega$ is a co-safety language, then all $w \in L$ have a finite good prefix.
- Let L be co-safety, then $\overline{L}$ is safety.
- $p\mathbf{U}q$ is co-safety
- $\mathbf{F}p$ is co-safety (but also liveness)

Is that all? (Cont'd)

Other

- There are properties which are both, safety and co-safety, or co-safety and liveness, etc. We call them "other".



- Natural question to ask: "which properties are the *monitorable properties*, MON?" (cf. [PZ06])

The SSL Protocol

Some facts in a nutshell

- Secure Sockets Layer: Cryptographic protocol providing secure communication on the Internet
- In the protocol stack, between higher-level protocols (HTTP, FTP, etc.) and TCP/IP layer
  - as such, can also exist in user-space
- Many implementations exist (OpenSSL, Jessie, etc.)
- Most common attack: Man-in-the-middle-attack, trying to intercept, block, and alter messages
  - Typically, attacker has to interfere with the handshake phase of protocol, when certificates are exchanged
- Other attacks: E.g., attack cryptohashing functions for MAC-address comparison, etc.

The SSL Protocol

Some facts in a nutshell

- Secure Sockets Layer: Cryptographic protocol providing secure communication on the Internet
- In the protocol stack, between higher-level protocols (HTTP, FTP, etc.) and TCP/IP layer
  - as such, can also exist in user-space
- Many implementations exist (OpenSSL, Jessie, etc.)
- Most common attack: Man-in-the-middle-attack, trying to intercept, block, and alter messages
  - Typically, attacker has to interfere with the handshake phase of protocol, when certificates are exchanged
- Other attacks: E.g., attack cryptohashing functions for MAC-address comparison, etc.

The SSL Protocol

Some facts in a nutshell

- Secure Sockets Layer: Cryptographic protocol providing secure communication on the Internet
- In the protocol stack, between higher-level protocols (HTTP, FTP, etc.) and TCP/IP layer
  - as such, can also exist in user-space
- Many implementations exist (OpenSSL, Jessie, etc.)
- Most common attack: Man-in-the-middle-attack, trying to intercept, block, and alter messages
  - Typically, attacker has to interfere with the handshake phase of protocol, when certificates are exchanged
- Other attacks: E.g., attack cryptohashing functions for MAC-address comparison, etc.

The SSL Protocol

Some facts in a nutshell

- Secure Sockets Layer: Cryptographic protocol providing secure communication on the Internet
- In the protocol stack, between higher-level protocols (HTTP, FTP, etc.) and TCP/IP layer
  - as such, can also exist in user-space
- Many implementations exist (OpenSSL, Jessie, etc.)
- Most common attack: Man-in-the-middle-attack, trying to intercept, block, and alter messages
  - Typically, attacker has to interfere with the handshake phase of protocol, when certificates are exchanged
- Other attacks: E.g., attack cryptohashing functions for MAC-address comparison, etc.

The SSL Protocol

Some facts in a nutshell

- Secure Sockets Layer: Cryptographic protocol providing secure communication on the Internet
- In the protocol stack, between higher-level protocols (HTTP, FTP, etc.) and TCP/IP layer
  - as such, can also exist in user-space
- Many implementations exist (OpenSSL, Jessie, etc.)
- Most common attack: Man-in-the-middle-attack, trying to intercept, block, and alter messages
  - Typically, attacker has to interfere with the handshake phase of protocol, when certificates are exchanged
- Other attacks: E.g., attack cryptohashing functions for MAC-address comparison, etc.

The SSL Protocol

Some facts in a nutshell

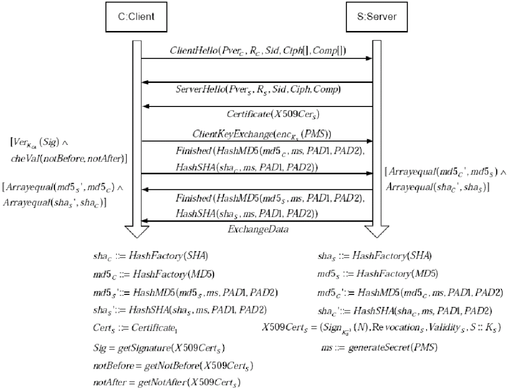
- Secure Sockets Layer: Cryptographic protocol providing secure communication on the Internet
- In the protocol stack, between higher-level protocols (HTTP, FTP, etc.) and TCP/IP layer
  - as such, can also exist in user-space
- Many implementations exist (OpenSSL, Jessie, etc.)
- Most common attack: Man-in-the-middle-attack, trying to intercept, block, and alter messages
  - Typically, attacker has to interfere with the handshake phase of protocol, when certificates are exchanged
- Other attacks: E.g., attack cryptohashing functions for MAC-address comparison, etc.

The SSL Protocol

Some facts in a nutshell

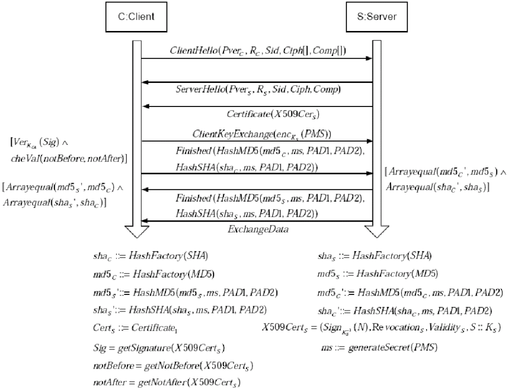
- Secure Sockets Layer: Cryptographic protocol providing secure communication on the Internet
- In the protocol stack, between higher-level protocols (HTTP, FTP, etc.) and TCP/IP layer
  - as such, can also exist in user-space
- Many implementations exist (OpenSSL, Jessie, etc.)
- Most common attack: Man-in-the-middle-attack, trying to intercept, block, and alter messages
  - Typically, attacker has to interfere with the handshake phase of protocol, when certificates are exchanged
- Other attacks: E.g., attack cryptohashing functions for MAC-address comparison, etc.

Monitoring the SSL handshake



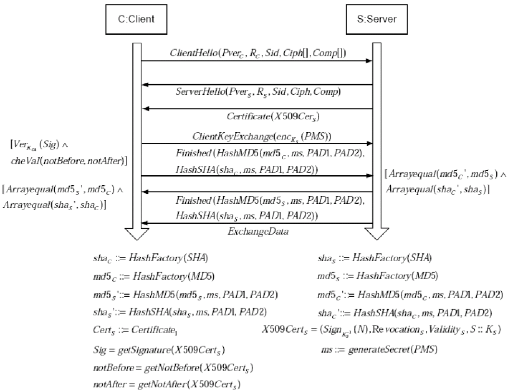
- Instead of generating behavioural model, we extract LTL properties directly from the model and/or already formalised FOL-security properties
- FOL over words and LTL expressively equivalent [Ka68]

Monitoring the SSL handshake



- Instead of generating behavioural model, we extract LTL properties directly from the model and/or already formalised FOL-security properties
- FOL over words and LTL expressively equivalent [Ka68]

Monitoring the SSL handshake



- Instead of generating behavioural model, we extract LTL properties directly from the model and/or already formalised FOL-security properties
- FOL over words and LTL expressively equivalent [Ka68]

LTL security properties of the SSL protocol

Security property 1

"Client won't send out ClientKeyExchange(enc_K, (PMS)) until it has received Certificate($X509Cer_S$), and the validity check of the certificate is positive."

To specify this in LTL, we have to

1. define alphabet accordingly wrt. abstract functions & messages
2. identify which functions & messages are relevant
3. instrument code to transmit abstract values to monitor

LTL security properties of the SSL protocol

Security property 1

"Client won't send out ClientKeyExchange($enc_K, (PMS)$) until it has received Certificate($X509Cer_S$), and the validity check of the certificate is positive."

To specify this in LTL, we have to

1. define alphabet accordingly wrt. abstract functions & messages
2. identify which functions & messages are relevant
3. instrument code to transmit abstract values to monitor

LTL security properties of the SSL protocol

Security property 1

"Client won't send out ClientKeyExchange($enc_K, (PMS)$) until it has received Certificate($X509Cer_S$), and the validity check of the certificate is positive."

To specify this in LTL, we have to

1. define alphabet accordingly wrt. abstract functions & messages
2. identify which functions & messages are relevant
3. instrument code to transmit abstract values to monitor

LTL security properties of the SSL protocol

Security property 1

"Client won't send out ClientKeyExchange(enc_K, (PMS)) until it has received Certificate($X509Cer_S$), and the validity check of the certificate is positive."

To specify this in LTL, we have to

1. define alphabet accordingly wrt. abstract functions & messages
2. identify which functions & messages are relevant
3. instrument code to transmit abstract values to monitor

LTL security properties of the SSL protocol

Security property 1

"Client won't send out ClientKeyExchange(enc_K, (PMS)) until it has received Certificate($X509Cer_S$), and the validity check of the certificate is positive."

To specify this in LTL, we have to

1. define alphabet accordingly wrt. abstract functions & messages
2. identify which functions & messages are relevant
3. instrument code to transmit abstract values to monitor

LTL security properties of the SSL protocol

Security property 1

"Client won't send out ClientKeyExchange(enc_K, (PMS)) until it has received Certificate($X509Cer_S$), and the validity check of the certificate is positive."

To specify this in LTL, we have to

1. define alphabet accordingly wrt. abstract functions & messages
2. identify which functions & messages are relevant
3. instrument code to transmit abstract values to monitor

in Model	Send: ClientHello	by Outputstream.write in
	type.getValue()	Handshake.write
	(bout.size() >>> 16 & 0xFF)	Handshake.write
	(bout.size() >>> 8 & 0xFF)	Handshake.write
	(bout.size() & 0xFF)	Handshake.write
Pver	major	ProtocolVersion.write
	minor	ProtocolVersion.write
	((gmtUnixTime >>> 24) & 0xFF)	Random.write
	((gmtUnixTime >>> 16) & 0xFF)	Random.write
	((gmtUnixTime >>> 8) & 0xFF)	Random.write
	(gmtUnixTime & 0xFF)	Random.write
R_C	randomBytes	ClientHello.write
	sessionId.length	ClientHello.write
Sid	sessionId	ClientHello.write
	((suites.size() << 1) >>> 8 & 0xFF)	ClientHello.write
	((suites.size() << 1) & 0xFF)	ClientHello.write
Ciph[]	id[]	CipherSuite.write
	comp.size()	ClientHello.write
Comp[]	comp[2]	ClientHello.write

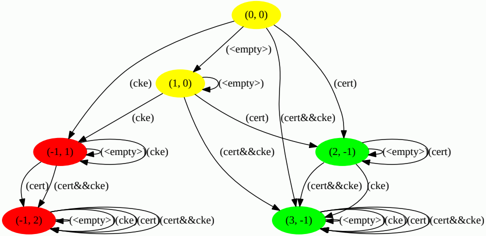
LTL security properties of the SSL protocol (Cont'd)

Security property 1 in LTL

- $\varphi_1 = \neg\text{ClientKeyExchange}(enc_K, (PMS))\mathbf{U}_w\text{Certificate}(X509Cer_S)$
- Safety property

Monitor for φ_1

- Finite state machine of Moore-type:



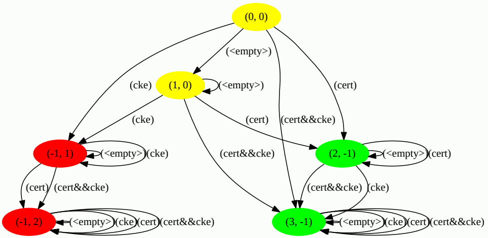
LTL security properties of the SSL protocol (Cont'd)

Security property 1 in LTL

- $\varphi_1 = \neg\text{ClientKeyExchange}(enc_K, (PMS))\mathbf{U}_w\text{Certificate}(X509Cer_S)$
- Safety property

Monitor for φ_1

- Finite state machine of Moore-type:



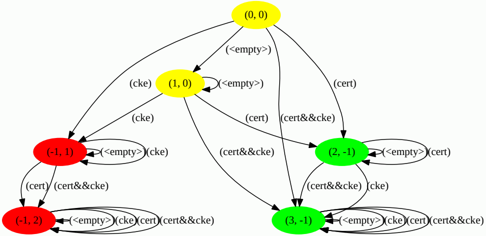
LTL security properties of the SSL protocol (Cont'd)

Security property 1 in LTL

- $\varphi_1 = \neg\text{ClientKeyExchange}(enc_K, (PMS))\mathbf{U}_w\text{Certificate}(X509Cer_S)$
- Safety property

Monitor for φ_1

- Finite state machine of Moore-type:



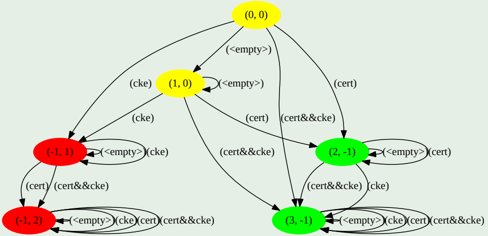
LTL security properties of the SSL protocol (Cont'd)

Security property 1 in LTL

- $\varphi_1 = \neg\text{ClientKeyExchange}(enc_K, (PMS))\mathbf{U}_w\text{Certificate}(X509Cer_S)$
- Safety property

Monitor for φ_1

- Finite state machine of Moore-type:



LTL security properties of the SSL protocol (Cont'd)

Security property 2

$\mathrm{Finished}(HashMD5(md5_s, ms, PAD1, PAD2))$ is not sent by the server to the client before the MD5 hash received from the client in the message $\mathrm{Finished}(HashMD5(md5_c, ms, PAD1, PAD2))$ has been checked to be equal to the MD5 created by the server, and correspondingly for the SHA hash, but will send it out eventually after that has been established.

Security property 2 in LTL

$$\begin{aligned}\varphi_2 = \quad & (\neg \mathrm{Finished}(HashMD5(md5_s, ms, PAD1, PAD2)) \\ & \quad \mathbf{U}_w \mathrm{Arrayequal}(md5_s, md5_c)) \\ & \quad\quad \wedge (\mathbf{F}\mathrm{Arrayequal}(md5_s, md5_c) \\ & \quad\quad\quad \Rightarrow \mathbf{F}\mathrm{Finished}(HashMD5(md5_s, ms, \ldots))).\end{aligned}$$

- Not co-safety, not safety

LTL security properties of the SSL protocol (Cont'd)

Security property 2

Finished($HashMD5(md5_s, ms, PAD1, PAD2)$) is not sent by the server to the client before the MD5 hash received from the client in the message Finished($HashMD5(md5_c, ms, PAD1, PAD2)$) has been checked to be equal to the MD5 created by the server, and correspondingly for the SHA hash, but will send it out eventually after that has been established.

Security property 2 in LTL

$$\begin{aligned}\varphi_2 = \quad & (\neg \text{Finished}(HashMD5(md5_s, ms, PAD1, PAD2)) \\ & \quad \mathbf{U}_w \text{Arrayequal}(md5_s, md5_c)) \\ & \quad\quad \wedge(\mathbf{F}\text{Arrayequal}(md5_s, md5_c) \\ & \quad\quad\quad \Rightarrow \mathbf{F}\text{Finished}(HashMD5(md5_s, ms, \ldots))).\end{aligned}$$

- Not co-safety, not safety

LTL security properties of the SSL protocol (Cont'd)

Security property 2

Finished($HashMD5(md5_s, ms, PAD1, PAD2)$) is not sent by the server to the client before the MD5 hash received from the client in the message Finished($HashMD5(md5_c, ms, PAD1, PAD2)$) has been checked to be equal to the MD5 created by the server, and correspondingly for the SHA hash, but will send it out eventually after that has been established.

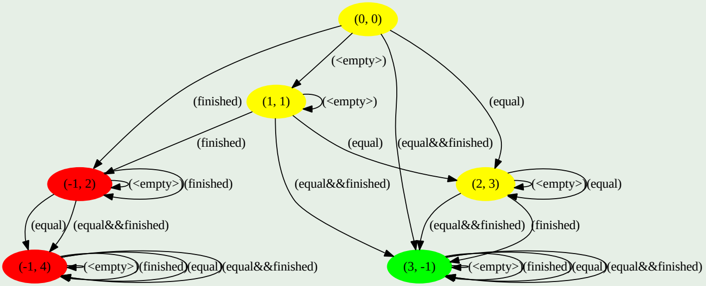
Security property 2 in LTL

$$\varphi_2 = (\neg\text{Finished}(HashMD5(md5_s, ms, PAD1, PAD2)) \; \mathbf{U}_w \text{Arrayequal}(md5_s, md5_c)) \wedge (\mathbf{F}\text{Arrayequal}(md5_s, md5_c) \Rightarrow \mathbf{F}\text{Finished}(HashMD5(md5_s, ms, \ldots))).$$

- Not co-safety, not safety

LTL security properties of the SSL protocol (Cont'd)

Monitor for φ_2



LTL security properties of the SSL protocol (Cont'd)

Security property 3

"The client will not send any transport data to the server before the MD5 hash received from the server in the Finished message has been checked to be equal to the MD5 created by the client, and correspondingly for the SHA hash."

Security property 3 in LTL

- $\varphi_3 = \neg Data \mathbf{U}_w ((MD5(\text{Finished}_R) = MD5(\text{Finished}_S))$
- Safety

LTL security properties of the SSL protocol (Cont'd)

Security property 3

"The client will not send any transport data to the server before the MD5 hash received from the server in the Finished message has been checked to be equal to the MD5 created by the client, and correspondingly for the SHA hash."

Security property 3 in LTL

- $\varphi_3 = \neg Data \mathbf{U}_w((MD5(\text{Finished}_R) = MD5(\text{Finished}_S))$
- Safety

LTL security properties of the SSL protocol (Cont'd)

Security property 3

"The client will not send any transport data to the server before the MD5 hash received from the server in the Finished message has been checked to be equal to the MD5 created by the client, and correspondingly for the SHA hash."

Security property 3 in LTL

- $\varphi_3 = \neg Data \mathbf{U}_w((MD5(\text{Finished}_R) = MD5(\text{Finished}_S))$
- Safety

Runtime verification vs. model checking

LTL model checking using Büchi automata:

- Translation: $\varphi \mapsto \mathcal{A}^{\varphi}$ s. t. $\mathcal{L}(\mathcal{A}^{\varphi})$ = models of φ
- $S \models \varphi$: every run in S satisfies φ, i. e., $\mathcal{L}(S \times \mathcal{A}^{\neg\varphi}) = \emptyset$?
- Language inclusion often of higher complexity than "word problem", i. e. $s \in \mathcal{L}(\varphi)$?

Runtime verification vs. model checking

LTL model checking using Büchi automata:

- Translation: $\varphi \mapsto \mathcal{A}^{\varphi}$ s. t. $\mathcal{L}(\mathcal{A}^{\varphi})$ = models of φ
- $S \models \varphi$: every run in S satisfies φ, i. e., $\mathcal{L}(S \times \mathcal{A}^{\neg\varphi}) = \emptyset$?
- Language inclusion often of higher complexity than "word problem", i. e. $s \in \mathcal{L}(\varphi)$?

Runtime verification vs. model checking

LTL model checking using Büchi automata:

- Translation: $\varphi \mapsto \mathcal{A}^\varphi$ s. t. $\mathcal{L}(\mathcal{A}^\varphi) =$ models of φ
- $S \models \varphi$: every run in S satisfies φ, i. e., $\mathcal{L}(S \times \mathcal{A}^{\neg\varphi}) = \emptyset$?
- Language inclusion often of higher complexity than "word problem", i. e. $s \in \mathcal{L}(\varphi)$?

Runtime verification vs. model checking

LTL model checking using Büchi automata:

- Translation: $\varphi \mapsto \mathcal{A}^{\varphi}$ s. t. $\mathcal{L}(\mathcal{A}^{\varphi}) =$ models of φ
- $S \models \varphi$: every run in S satisfies φ, i. e., $\mathcal{L}(S \times \mathcal{A}^{\neg\varphi}) = \emptyset$?
- Language inclusion often of higher complexity than "word problem", i. e. $s \in \mathcal{L}(\varphi)$?

Properties of Büchi automata

- Büchi automata are nondeterministic (determinisation possible but exponential time lower bound [Saf89])
- Büchi-acceptance defined over infinite words
  - Runtime verification deals only with prefixes
  - LTL semantics defined over infinite words, e. g., how to interpret $\mathbf{X}p$ at the end of a trace?
- For many applications, synchronous notion of time not sufficient $\Rightarrow$ support for timed languages

Runtime verification vs. model checking

LTL model checking using Büchi automata:

- Translation: $\varphi \mapsto \mathcal{A}^\varphi$ s. t. $\mathcal{L}(\mathcal{A}^\varphi) =$ models of φ
- $S \models \varphi$: every run in S satisfies φ, i. e., $\mathcal{L}(S \times \mathcal{A}^{\neg\varphi}) = \emptyset$?
- Language inclusion often of higher complexity than "word problem", i. e. $s \in \mathcal{L}(\varphi)$?

Properties of Büchi automata

- Büchi automata are nondeterministic (determinisation possible but exponential time lower bound [Saf89])
- Büchi-acceptance defined over infinite words
  - Runtime verification deals only with prefixes
  - LTL semantics defined over infinite words, e. g., how to interpret $\mathbf{X}p$ at the end of a trace?
- For many applications, synchronous notion of time not sufficient $\Rightarrow$ support for timed languages

Runtime verification vs. model checking

LTL model checking using Büchi automata:

- Translation: $\varphi \mapsto \mathcal{A}^\varphi$ s. t. $\mathcal{L}(\mathcal{A}^\varphi)$ = models of φ
- $S \models \varphi$: every run in S satisfies φ, i. e., $\mathcal{L}(S \times \mathcal{A}^{\neg\varphi}) = \emptyset$?
- Language inclusion often of higher complexity than "word problem", i. e. $s \in \mathcal{L}(\varphi)$?

Properties of Büchi automata

- Büchi automata are nondeterministic (determinisation possible but exponential time lower bound [Saf89])
- Büchi-acceptance defined over infinite words
  - Runtime verification deals only with prefixes
  - LTL semantics defined over infinite words, e. g., how to interpret $\mathbf{X}p$ at the end of a trace?
- For many applications, synchronous notion of time not sufficient $\Rightarrow$ support for timed languages

An extension semantics for LTL

Definition: Traditional LTL semantics

Given $w \in \Sigma^\omega$, $\varphi \in \text{LTL}$, then $w \models \varphi \in \{\top, \bot\}$

Definition: Extension semantics over $\{\top, \bot, ?\}$: LTL_3

Given $u \in \Sigma^*$, then

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

An extension semantics for LTL

Definition: Traditional LTL semantics

Given $w \in \Sigma^\omega$, $\varphi \in \mathrm{LTL}$, then $w \models \varphi \in \{\top, \bot\}$

Definition: Extension semantics over $\{\top, \bot, ?\}$: LTL_3

Given $u \in \Sigma^*$, then

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

An extension semantics for LTL

Definition: Traditional LTL semantics

Given $w \in \Sigma^\omega$, $\varphi \in \text{LTL}$, then $w \models \varphi \in \{\top, \bot\}$

Definition: Extension semantics over $\{\top, \bot, ?\}$: LTL$_3$

Given $u \in \Sigma^*$, then

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

An extension semantics for LTL

Definition: Traditional LTL semantics

Given $w \in \Sigma^\omega$, $\varphi \in$ LTL, then $w \models \varphi \in \{\top, \bot\}$

Definition: Extension semantics over $\{\top, \bot, ?\}$: LTL_3

Given $u \in \Sigma^*$, then

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

Wanted: An on-the-fly decision procedure for LTL_3

- How can we determine ?, i. e., whether $\exists w \in \Sigma^\omega : uw \models \varphi$ and $\exists w' \in \Sigma^\omega : uw' \not\models \varphi$?
- How can we do this efficiently, i. e., at runtime?

An extension semantics for LTL

Definition: Traditional LTL semantics

Given $w \in \Sigma^\omega$, $\varphi \in$ LTL, then $w \models \varphi \in \{\top, \bot\}$

Definition: Extension semantics over $\{\top, \bot, ?\}$: LTL$_3$

Given $u \in \Sigma^*$, then

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

Wanted: An on-the-fly decision procedure for LTL$_3$

- How can we determine ?, i. e., whether $\exists w \in \Sigma^\omega : uw \models \varphi$ and $\exists w' \in \Sigma^\omega : uw' \not\models \varphi$?
- How can we do this efficiently, i. e., at runtime?

Towards an on-the-fly decision procedure for LTL_3

1. Translation: $\varphi \mapsto \mathcal{A}^{\varphi}$, s. t. $\mathcal{L}(\mathcal{A}^{\varphi}) = \mathcal{L}(\varphi)$
2. Emptiness per state: Labelling $\mathcal{F} : Q^{\varphi} \rightarrow \{\top, \bot\}$
3. NFA: Turn $\mathcal{A}^{\varphi}$ into NFA $\hat{\mathcal{A}}^{\varphi}$ using $\mathcal{F}$ as accepting states
4. DFA: Determinise $\hat{\mathcal{A}}^{\varphi}$

Towards an on-the-fly decision procedure for LTL_3

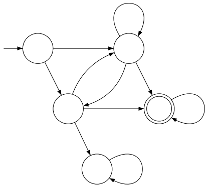
1. Translation: $\varphi \mapsto \mathcal{A}^\varphi$, s. t. $\mathcal{L}(\mathcal{A}^\varphi) = \mathcal{L}(\varphi)$
2. Emptiness per state: Labelling $\mathcal{F} : Q^\varphi \to \{\top, \bot\}$
3. NFA: Turn $\mathcal{A}^\varphi$ into NFA $\hat{\mathcal{A}}^\varphi$ using $\mathcal{F}$ as accepting states
4. DFA: Determinise $\hat{\mathcal{A}}^\varphi$

Towards an on-the-fly decision procedure for LTL_3

1. Translation: $\varphi \mapsto \mathcal{A}^\varphi$, s. t. $\mathcal{L}(\mathcal{A}^\varphi) = \mathcal{L}(\varphi)$
2. Emptiness per state: Labelling $\mathcal{F} : Q^\varphi \rightarrow \{\top, \bot\}$
3. NFA: Turn $\mathcal{A}^\varphi$ into NFA $\hat{\mathcal{A}}^\varphi$ using $\mathcal{F}$ as accepting states
4. DFA: Determinise $\hat{\mathcal{A}}^\varphi$

Towards an on-the-fly decision procedure for LTL_3

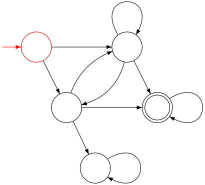
1. Translation: $\varphi \mapsto \mathcal{A}^\varphi$, s. t. $\mathcal{L}(\mathcal{A}^\varphi) = \mathcal{L}(\varphi)$
2. Emptiness per state: Labelling $\mathcal{F} : Q^\varphi \to \{\top, \bot\}$



3. NFA: Turn $\mathcal{A}^\varphi$ into NFA $\hat{\mathcal{A}}^\varphi$ using $\mathcal{F}$ as accepting states
4. DFA: Determinise $\hat{\mathcal{A}}^\varphi$

Towards an on-the-fly decision procedure for LTL_3

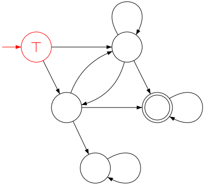
1. Translation: $\varphi \mapsto \mathcal{A}^\varphi$, s. t. $\mathcal{L}(\mathcal{A}^\varphi) = \mathcal{L}(\varphi)$
2. Emptiness per state: Labelling $\mathcal{F} : Q^\varphi \to \{\top, \bot\}$



3. NFA: Turn $\mathcal{A}^\varphi$ into NFA $\hat{\mathcal{A}}^\varphi$ using $\mathcal{F}$ as accepting states
4. DFA: Determinise $\hat{\mathcal{A}}^\varphi$

Towards an on-the-fly decision procedure for LTL_3

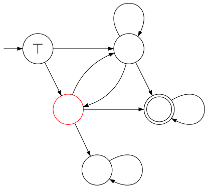
1. Translation: $\varphi \mapsto \mathcal{A}^\varphi$, s. t. $\mathcal{L}(\mathcal{A}^\varphi) = \mathcal{L}(\varphi)$
2. Emptiness per state: Labelling $\mathcal{F} : Q^\varphi \to \{\top, \bot\}$



3. NFA: Turn $\mathcal{A}^\varphi$ into NFA $\hat{\mathcal{A}}^\varphi$ using $\mathcal{F}$ as accepting states
4. DFA: Determinise $\hat{\mathcal{A}}^\varphi$

Towards an on-the-fly decision procedure for LTL_3

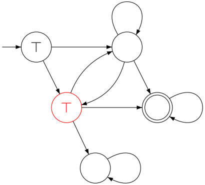
1. Translation: $\varphi \mapsto \mathcal{A}^\varphi$, s. t. $\mathcal{L}(\mathcal{A}^\varphi) = \mathcal{L}(\varphi)$
2. Emptiness per state: Labelling $\mathcal{F} : Q^\varphi \rightarrow \{\top, \bot\}$



3. NFA: Turn $\mathcal{A}^\varphi$ into NFA $\hat{\mathcal{A}}^\varphi$ using $\mathcal{F}$ as accepting states
4. DFA: Determinise $\hat{\mathcal{A}}^\varphi$

Towards an on-the-fly decision procedure for LTL_3

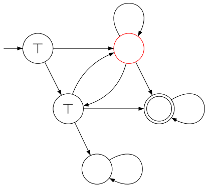
1. Translation: $\varphi \mapsto \mathcal{A}^\varphi$, s. t. $\mathcal{L}(\mathcal{A}^\varphi) = \mathcal{L}(\varphi)$
2. Emptiness per state: Labelling $\mathcal{F} : Q^\varphi \to \{\top, \bot\}$



3. NFA: Turn $\mathcal{A}^\varphi$ into NFA $\hat{\mathcal{A}}^\varphi$ using $\mathcal{F}$ as accepting states
4. DFA: Determinise $\hat{\mathcal{A}}^\varphi$

Towards an on-the-fly decision procedure for LTL_3

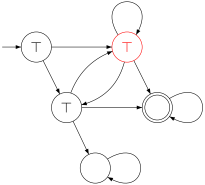
1. Translation: $\varphi \mapsto \mathcal{A}^\varphi$, s. t. $\mathcal{L}(\mathcal{A}^\varphi) = \mathcal{L}(\varphi)$
2. Emptiness per state: Labelling $\mathcal{F} : Q^\varphi \rightarrow \{\top, \bot\}$



3. NFA: Turn $\mathcal{A}^\varphi$ into NFA $\hat{\mathcal{A}}^\varphi$ using $\mathcal{F}$ as accepting states
4. DFA: Determinise $\hat{\mathcal{A}}^\varphi$

Towards an on-the-fly decision procedure for LTL_3

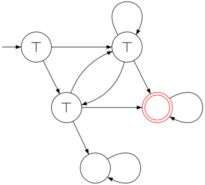
1. Translation: $\varphi \mapsto \mathcal{A}^\varphi$, s. t. $\mathcal{L}(\mathcal{A}^\varphi) = \mathcal{L}(\varphi)$
2. Emptiness per state: Labelling $\mathcal{F} : Q^\varphi \to \{\top, \bot\}$



3. NFA: Turn $\mathcal{A}^\varphi$ into NFA $\hat{\mathcal{A}}^\varphi$ using $\mathcal{F}$ as accepting states
4. DFA: Determinise $\hat{\mathcal{A}}^\varphi$

Towards an on-the-fly decision procedure for LTL_3

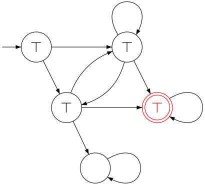
1. Translation: $\varphi \mapsto \mathcal{A}^\varphi$, s. t. $\mathcal{L}(\mathcal{A}^\varphi) = \mathcal{L}(\varphi)$
2. Emptiness per state: Labelling $\mathcal{F} : Q^\varphi \to \{\top, \bot\}$



3. NFA: Turn $\mathcal{A}^\varphi$ into NFA $\hat{\mathcal{A}}^\varphi$ using $\mathcal{F}$ as accepting states
4. DFA: Determinise $\hat{\mathcal{A}}^\varphi$

Towards an on-the-fly decision procedure for LTL_3

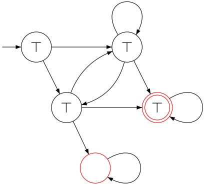
1. Translation: $\varphi \mapsto \mathcal{A}^\varphi$, s. t. $\mathcal{L}(\mathcal{A}^\varphi) = \mathcal{L}(\varphi)$
2. Emptiness per state: Labelling $\mathcal{F} : Q^\varphi \to \{\top, \bot\}$



3. NFA: Turn $\mathcal{A}^\varphi$ into NFA $\hat{\mathcal{A}}^\varphi$ using $\mathcal{F}$ as accepting states
4. DFA: Determinise $\hat{\mathcal{A}}^\varphi$

Towards an on-the-fly decision procedure for LTL_3

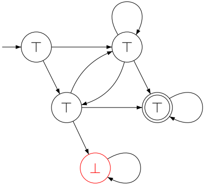
1. Translation: $\varphi \mapsto \mathcal{A}^\varphi$, s. t. $\mathcal{L}(\mathcal{A}^\varphi) = \mathcal{L}(\varphi)$
2. Emptiness per state: Labelling $\mathcal{F} : Q^\varphi \to \{\top, \bot\}$



3. NFA: Turn $\mathcal{A}^\varphi$ into NFA $\hat{\mathcal{A}}^\varphi$ using $\mathcal{F}$ as accepting states
4. DFA: Determinise $\hat{\mathcal{A}}^\varphi$

Towards an on-the-fly decision procedure for LTL_3

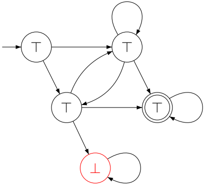
1. Translation: $\varphi \mapsto \mathcal{A}^\varphi$, s. t. $\mathcal{L}(\mathcal{A}^\varphi) = \mathcal{L}(\varphi)$
2. Emptiness per state: Labelling $\mathcal{F} : Q^\varphi \to \{\top, \bot\}$



3. NFA: Turn $\mathcal{A}^\varphi$ into NFA $\hat{\mathcal{A}}^\varphi$ using $\mathcal{F}$ as accepting states
4. DFA: Determinise $\hat{\mathcal{A}}^\varphi$

Towards an on-the-fly decision procedure for LTL_3

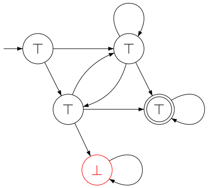
1. Translation: $\varphi \mapsto \mathcal{A}^\varphi$, s. t. $\mathcal{L}(\mathcal{A}^\varphi) = \mathcal{L}(\varphi)$
2. Emptiness per state: Labelling $\mathcal{F} : Q^\varphi \rightarrow \{\top, \bot\}$



3. NFA: Turn $\mathcal{A}^\varphi$ into NFA $\hat{\mathcal{A}}^\varphi$ using $\mathcal{F}$ as accepting states
4. DFA: Determinise $\hat{\mathcal{A}}^\varphi$

Towards an on-the-fly decision procedure for LTL_3

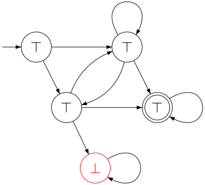
1. Translation: $\varphi \mapsto \mathcal{A}^\varphi$, s. t. $\mathcal{L}(\mathcal{A}^\varphi) = \mathcal{L}(\varphi)$
2. Emptiness per state: Labelling $\mathcal{F} : Q^\varphi \to \{\top, \bot\}$



3. NFA: Turn $\mathcal{A}^\varphi$ into NFA $\hat{\mathcal{A}}^\varphi$ using $\mathcal{F}$ as accepting states
4. DFA: Determinise $\hat{\mathcal{A}}^\varphi$

Towards an on-the-fly decision procedure for LTL_3

1. Translation: $\varphi \mapsto \mathcal{A}^\varphi$, s. t. $\mathcal{L}(\mathcal{A}^\varphi) = \mathcal{L}(\varphi)$
2. Emptiness per state: Labelling $\mathcal{F} : Q^\varphi \rightarrow \{\top, \bot\}$



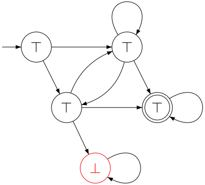
3. NFA: Turn $\mathcal{A}^\varphi$ into NFA $\hat{\mathcal{A}}^\varphi$ using $\mathcal{F}$ as accepting states
4. DFA: Determinise $\hat{\mathcal{A}}^\varphi$

Theorem

$$u \in \mathcal{L}(\hat{\mathcal{A}}^\varphi) \Leftrightarrow \varphi \textit{ is satisfiable}$$

Towards an on-the-fly decision procedure for LTL_3

1. Translation: $\varphi \mapsto \mathcal{A}^\varphi$, s. t. $\mathcal{L}(\mathcal{A}^\varphi) = \mathcal{L}(\varphi)$
2. Emptiness per state: Labelling $\mathcal{F} : Q^\varphi \rightarrow \{\top, \bot\}$



3. NFA: Turn $\mathcal{A}^\varphi$ into NFA $\hat{\mathcal{A}}^\varphi$ using $\mathcal{F}$ as accepting states
4. DFA: Determinise $\hat{\mathcal{A}}^\varphi$

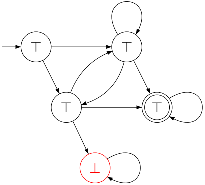
Theorem

$u \in \mathcal{L}(\hat{\mathcal{A}}^\varphi) \Leftrightarrow \varphi$ is satisfiable

However, there exists $u \in \mathcal{L}(\hat{\mathcal{A}}^\varphi)$, s. t. φ is unsatisfiable!

Towards an on-the-fly decision procedure for LTL_3

1. Translation: $\varphi \mapsto \mathcal{A}^\varphi$, s. t. $\mathcal{L}(\mathcal{A}^\varphi) = \mathcal{L}(\varphi)$
2. Emptiness per state: Labelling $\mathcal{F} : Q^\varphi \to \{\top, \bot\}$



3. NFA: Turn $\mathcal{A}^\varphi$ into NFA $\hat{\mathcal{A}}^\varphi$ using $\mathcal{F}$ as accepting states
4. DFA: Determinise $\hat{\mathcal{A}}^\varphi$

Theorem

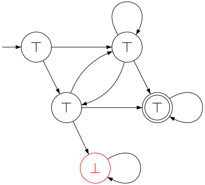
$u \in \mathcal{L}(\hat{\mathcal{A}}^\varphi) \Leftrightarrow \varphi$ *is satisfiable*

However, there exists $u \in \mathcal{L}(\hat{\mathcal{A}}^\varphi)$, s. t. φ is unsatisfiable!

1. Translation: $\neg\varphi \mapsto \mathcal{A}^{\neg\varphi}$, s. t. $\mathcal{L}(\mathcal{A}^{\neg\varphi}) = \Sigma^\omega \backslash \mathcal{L}(\varphi)$
2. Emptiness per state
3. NFA: Turn $\mathcal{A}^{\neg\varphi}$ into NFA
4. DFA: Determinise NFA

Towards an on-the-fly decision procedure for LTL_3

1. Translation: $\varphi \mapsto \mathcal{A}^{\varphi}$, s. t. $\mathcal{L}(\mathcal{A}^{\varphi}) = \mathcal{L}(\varphi)$
2. Emptiness per state: Labelling $\mathcal{F} : Q^{\varphi} \rightarrow \{\top, \bot\}$



3. NFA: Turn $\mathcal{A}^{\varphi}$ into NFA $\hat{\mathcal{A}}^{\varphi}$ using $\mathcal{F}$ as accepting states
4. DFA: Determinise $\hat{\mathcal{A}}^{\varphi}$

Theorem

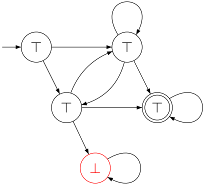
$u \in \mathcal{L}(\hat{\mathcal{A}}^{\varphi}) \Leftrightarrow \varphi$ is satisfiable

However, there exists $u \in \mathcal{L}(\hat{\mathcal{A}}^{\varphi})$, s. t. φ is unsatisfiable!

1. Translation: $\neg\varphi \mapsto \mathcal{A}^{\neg\varphi}$, s. t. $\mathcal{L}(\mathcal{A}^{\neg\varphi}) = \Sigma^{\omega} \backslash \mathcal{L}(\varphi)$
2. Emptiness per state
3. NFA: Turn $\mathcal{A}^{\neg\varphi}$ into NFA
4. DFA: Determinise NFA

Towards an on-the-fly decision procedure for LTL_3

1. Translation: $\varphi \mapsto \mathcal{A}^\varphi$, s. t. $\mathcal{L}(\mathcal{A}^\varphi) = \mathcal{L}(\varphi)$
2. Emptiness per state: Labelling $\mathcal{F} : Q^\varphi \to \{\top, \bot\}$



3. NFA: Turn $\mathcal{A}^\varphi$ into NFA $\hat{\mathcal{A}}^\varphi$ using $\mathcal{F}$ as accepting states
4. DFA: Determinise $\hat{\mathcal{A}}^\varphi$

Theorem

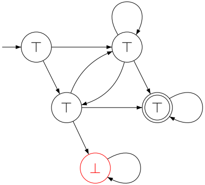
$u \in \mathcal{L}(\hat{\mathcal{A}}^\varphi) \Leftrightarrow \varphi$ *is satisfiable*

However, there exists $u \in \mathcal{L}(\hat{\mathcal{A}}^\varphi)$, s. t. φ is unsatisfiable!

1. Translation: $\neg\varphi \mapsto \mathcal{A}^{\neg\varphi}$, s. t. $\mathcal{L}(\mathcal{A}^{\neg\varphi}) = \Sigma^\omega \backslash \mathcal{L}(\varphi)$
2. Emptiness per state
3. NFA: Turn $\mathcal{A}^{\neg\varphi}$ into NFA
4. DFA: Determinise NFA

Towards an on-the-fly decision procedure for LTL_3

1. Translation: $\varphi \mapsto \mathcal{A}^\varphi$, s. t. $\mathcal{L}(\mathcal{A}^\varphi) = \mathcal{L}(\varphi)$
2. Emptiness per state: Labelling $\mathcal{F} : Q^\varphi \to \{\top, \bot\}$



3. NFA: Turn $\mathcal{A}^\varphi$ into NFA $\hat{\mathcal{A}}^\varphi$ using $\mathcal{F}$ as accepting states
4. DFA: Determinise $\hat{\mathcal{A}}^\varphi$

Theorem

$u \in \mathcal{L}(\hat{\mathcal{A}}^\varphi) \Leftrightarrow \varphi$ is satisfiable

However, there exists $u \in \mathcal{L}(\hat{\mathcal{A}}^\varphi)$, s. t. φ is unsatisfiable!

1. Translation: $\neg\varphi \mapsto \mathcal{A}^{\neg\varphi}$, s. t. $\mathcal{L}(\mathcal{A}^{\neg\varphi}) = \Sigma^\omega \backslash \mathcal{L}(\varphi)$
2. Emptiness per state
3. NFA: Turn $\mathcal{A}^{\neg\varphi}$ into NFA
4. DFA: Determinise NFA

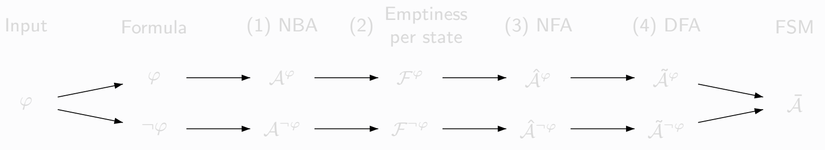
Monitor construction / decision procedure

Theorem

Given $u \in \Sigma^*$, $\varphi \in LTL$, then

$$[u \models \varphi] = \begin{cases} \top & \text{if } u \notin \mathcal{L}(\hat{\mathcal{A}}^{\neg\varphi}) \\ \bot & \text{if } u \notin \mathcal{L}(\hat{\mathcal{A}}^{\varphi}) \\ ? & \text{if } u \in \mathcal{L}(\hat{\mathcal{A}}^{\neg\varphi}) \text{ and } u \in \mathcal{L}(\hat{\mathcal{A}}^{\varphi}) \end{cases}$$

The procedure for getting $[u \models \varphi]$ for a given φ



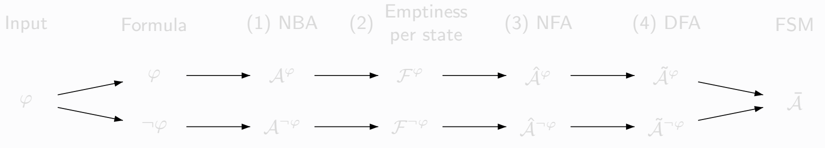
Monitor construction / decision procedure

Theorem

Given $u \in \Sigma^*$, $\varphi \in LTL$, *then*

$$[u \models \varphi] = \begin{cases} \top & \text{if } u \notin \mathcal{L}(\hat{\mathcal{A}}^{\neg\varphi}) \\ \bot & \text{if } u \notin \mathcal{L}(\hat{\mathcal{A}}^{\varphi}) \\ ? & \text{if } u \in \mathcal{L}(\hat{\mathcal{A}}^{\neg\varphi}) \text{ and } u \in \mathcal{L}(\hat{\mathcal{A}}^{\varphi}) \end{cases}$$

The procedure for getting $[u \models \varphi]$ for a given φ



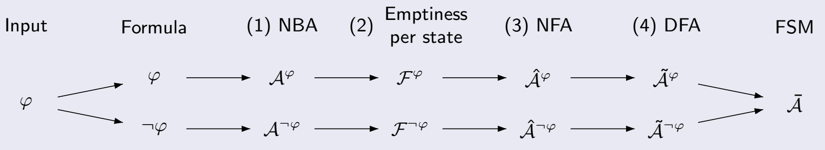
Monitor construction / decision procedure

Theorem

Given $u \in \Sigma^*$, $\varphi \in LTL$, *then*

$$[u \models \varphi] = \begin{cases} \top & \text{if } u \notin \mathcal{L}(\hat{\mathcal{A}}^{\neg\varphi}) \\ \bot & \text{if } u \notin \mathcal{L}(\hat{\mathcal{A}}^{\varphi}) \\ ? & \text{if } u \in \mathcal{L}(\hat{\mathcal{A}}^{\neg\varphi}) \text{ and } u \in \mathcal{L}(\hat{\mathcal{A}}^{\varphi}) \end{cases}$$

The procedure for getting $[u \models \varphi]$ for a given φ



Real-time

Timed words

$w \in T\Sigma^\omega := (a_0, t_0)(a_1, t_1)\ldots \quad (a_i \in \Sigma, t \in \mathbb{R}^{\geq 0})$

- Strict monotonicity: for each $i \in \mathbb{Z}$, $t_i < t_{i+1}$
- Progress: for all $t \in \mathbb{R}^{\geq 0}$ there is an $i \in \mathbb{N}$, s. t. $t_i > t$

(a_i, t_i) also called "event"

Timed languages

A timed language L is a set of timed words

- L is regular, if it is accepted by a timed automaton, whose language is L
- Kleene and McNaughton Theorems exist (but we do not care much right now. Active field of research.)

Real-time

Timed words

$w \in T\Sigma^\omega := (a_0, t_0)(a_1, t_1)\ldots \quad (a_i \in \Sigma, t \in \mathbb{R}^{\geq 0})$

- Strict monotonicity: for each $i \in \mathbb{Z}$, $t_i < t_{i+1}$
- Progress: for all $t \in \mathbb{R}^{\geq 0}$ there is an $i \in \mathbb{N}$, s. t. $t_i > t$

(a_i, t_i) also called "event"

Timed languages

A timed language L is a set of timed words

- L is regular, if it is accepted by a timed automaton, whose language is L
- Kleene and McNaughton Theorems exist (but we do not care much right now. Active field of research.)

Real-time

Timed words

$w \in T\Sigma^\omega := (a_0, t_0)(a_1, t_1)\dots \quad (a_i \in \Sigma, t \in \mathbb{R}^{\geq 0})$

- Strict monotonicity: for each $i \in \mathbb{Z}, t_i < t_{i+1}$
- Progress: for all $t \in \mathbb{R}^{\geq 0}$ there is an $i \in \mathbb{N}$, s. t. $t_i > t$

(a_i, t_i) also called "event"

Timed languages

A timed language L is a set of timed words

- L is regular, if it is accepted by a timed automaton, whose language is L
- Kleene and McNaughton Theorems exist (but we do not care much right now. Active field of research.)

Real-time

Timed words

$w \in T\Sigma^\omega := (a_0, t_0)(a_1, t_1)\dots \quad (a_i \in \Sigma, t \in \mathbb{R}^{\geq 0})$

- Strict monotonicity: for each $i \in \mathbb{Z}$, $t_i < t_{i+1}$
- Progress: for all $t \in \mathbb{R}^{\geq 0}$ there is an $i \in \mathbb{N}$, s. t. $t_i > t$

(a_i, t_i) also called "event"

Timed languages

A timed language L is a set of timed words

- L is regular, if it is accepted by a timed automaton, whose language is L
- Kleene and McNaughton Theorems exist (but we do not care much right now. Active field of research.)

Real-time

Timed words

$w \in T\Sigma^\omega := (a_0, t_0)(a_1, t_1)\dots \quad (a_i \in \Sigma, t \in \mathbb{R}^{\geq 0})$

- Strict monotonicity: for each $i \in \mathbb{Z}$, $t_i < t_{i+1}$
- Progress: for all $t \in \mathbb{R}^{\geq 0}$ there is an $i \in \mathbb{N}$, s. t. $t_i > t$

(a_i, t_i) also called "event"

Timed languages

A timed language L is a set of timed words

- L is regular, if it is accepted by a timed automaton, whose language is L
- Kleene and McNaughton Theorems exist (but we do not care much right now. Active field of research.)

Event clocks

For every $a \in \Sigma$, there exists a recording and a predicting clock to measure the distance between events.

Clock variables and valuations

$$\gamma_i(x_a) := \begin{cases} t_i - t_j & \text{if } \exists j < i : a_j = a \text{ and } \forall j < k < i : a_k \neq a \\ \perp & \text{otherwise} \end{cases}$$

$$\gamma_i(y_a) := \begin{cases} t_j - t_i & \text{if } \exists j > i : a_j = a \text{ and } \forall i < k < j : a_k \neq a \\ \perp & \text{otherwise} \end{cases}$$

Clock constraints

- Constraint: $z \bowtie c$, with $z \in C_\Sigma$, $c \in \mathbb{N}$, $\bowtie \in \{<, \leq, \geq, >\}$
- Example: $(x_a \leq 5) \in \Psi(C_\Sigma)$
- A valuation satisfies a constraint: $\gamma \models \psi \in \Psi(C_\Sigma)$
- Example: $\gamma(x_a) = 3.2 \models x_a \leq 5$

Event clocks

For every $a \in \Sigma$, there exists a recording and a predicting clock to measure the distance between events.

Clock variables and valuations

$$\gamma_i(x_a) := \begin{cases} t_i - t_j & \text{if } \exists j < i : a_j = a \text{ and } \forall j < k < i : a_k \neq a \\ \bot & \text{otherwise} \end{cases}$$

$$\gamma_i(y_a) := \begin{cases} t_j - t_i & \text{if } \exists j > i : a_j = a \text{ and } \forall i < k < j : a_k \neq a \\ \bot & \text{otherwise} \end{cases}$$

Clock constraints

- Constraint: $z \bowtie c$, with $z \in C_\Sigma$, $c \in \mathbb{N}$, $\bowtie \in \{<, \leq, \geq, >\}$
- Example: $(x_a \leq 5) \in \Psi(C_\Sigma)$
- A valuation satisfies a constraint: $\gamma \models \psi \in \Psi(C_\Sigma)$
- Example: $\gamma(x_a) = 3.2 \models x_a \leq 5$

Event clocks

For every $a \in \Sigma$, there exists a recording and a predicting clock to measure the distance between events.

Clock variables and valuations

$$\gamma_i(x_a) := \begin{cases} t_i - t_j & \text{if } \exists j < i : a_j = a \text{ and } \forall j < k < i : a_k \neq a \\ \bot & \text{otherwise} \end{cases}$$

$$\gamma_i(y_a) := \begin{cases} t_j - t_i & \text{if } \exists j > i : a_j = a \text{ and } \forall i < k < j : a_k \neq a \\ \bot & \text{otherwise} \end{cases}$$

Clock constraints

- Constraint: $z \bowtie c$, with $z \in C_\Sigma$, $c \in \mathbb{N}$, $\bowtie \in \{<, \leq, \geq, >\}$
- Example: $(x_a \leq 5) \in \Psi(C_\Sigma)$
- A valuation satisfies a constraint: $\gamma \models \psi \in \Psi(C_\Sigma)$
- Example: $\gamma(x_a) = 3.2 \models x_a \leq 5$

Event clocks

For every $a \in \Sigma$, there exists a recording and a predicting clock to measure the distance between events.

Clock variables and valuations

$$\gamma_i(x_a) := \begin{cases} t_i - t_j & \text{if } \exists j < i : a_j = a \text{ and } \forall j < k < i : a_k \neq a \\ \bot & \text{otherwise} \end{cases}$$

$$\gamma_i(y_a) := \begin{cases} t_j - t_i & \text{if } \exists j > i : a_j = a \text{ and } \forall i < k < j : a_k \neq a \\ \bot & \text{otherwise} \end{cases}$$

Clock constraints

- Constraint: $z \bowtie c$, with $z \in C_\Sigma$, $c \in \mathbb{N}$, $\bowtie \in \{<, \leq, \geq, >\}$
- Example: $(x_a \leq 5) \in \Psi(C_\Sigma)$
- A valuation satisfies a constraint: $\gamma \models \psi \in \Psi(C_\Sigma)$
- Example: $\gamma(x_a) = 3.2 \models x_a \leq 5$

Event clocks

For every $a \in \Sigma$, there exists a recording and a predicting clock to measure the distance between events.

Clock variables and valuations

$$\gamma_i(x_a) := \begin{cases} t_i - t_j & \text{if } \exists j < i : a_j = a \text{ and } \forall j < k < i : a_k \neq a \\ \perp & \text{otherwise} \end{cases}$$

$$\gamma_i(y_a) := \begin{cases} t_j - t_i & \text{if } \exists j > i : a_j = a \text{ and } \forall i < k < j : a_k \neq a \\ \perp & \text{otherwise} \end{cases}$$

Clock constraints

- Constraint: $z \bowtie c$, with $z \in C_\Sigma$, $c \in \mathbb{N}$, $\bowtie \in \{<, \leq, \geq, >\}$
- Example: $(x_a \leq 5) \in \Psi(C_\Sigma)$
- A valuation satisfies a constraint: $\gamma \models \psi \in \Psi(C_\Sigma)$
- Example: $\gamma(x_a) = 3.2 \models x_a \leq 5$

Event clocks

For every $a \in \Sigma$, there exists a recording and a predicting clock to measure the distance between events.

Clock variables and valuations

$$\gamma_i(x_a) := \begin{cases} t_i - t_j & \text{if } \exists j < i : a_j = a \text{ and } \forall j < k < i : a_k \neq a \\ \bot & \text{otherwise} \end{cases}$$

$$\gamma_i(y_a) := \begin{cases} t_j - t_i & \text{if } \exists j > i : a_j = a \text{ and } \forall i < k < j : a_k \neq a \\ \bot & \text{otherwise} \end{cases}$$

Clock constraints

- Constraint: $z \bowtie c$, with $z \in C_\Sigma$, $c \in \mathbb{N}$, $\bowtie \in \{<, \leq, \geq, >\}$
- Example: $(x_a \leq 5) \in \Psi(C_\Sigma)$
- A valuation satisfies a constraint: $\gamma \models \psi \in \Psi(C_\Sigma)$
- Example: $\gamma(x_a) = 3.2 \models x_a \leq 5$

Event clocks

For every $a \in \Sigma$, there exists a recording and a predicting clock to measure the distance between events.

Clock variables and valuations

$$\gamma_i(x_a) := \begin{cases} t_i - t_j & \text{if } \exists j < i : a_j = a \text{ and } \forall j < k < i : a_k \neq a \\ \bot & \text{otherwise} \end{cases}$$

$$\gamma_i(y_a) := \begin{cases} t_j - t_i & \text{if } \exists j > i : a_j = a \text{ and } \forall i < k < j : a_k \neq a \\ \bot & \text{otherwise} \end{cases}$$

Clock constraints

- Constraint: $z \bowtie c$, with $z \in C_\Sigma$, $c \in \mathbb{N}$, $\bowtie \in \{<, \leq, \geq, >\}$
- Example: $(x_a \leq 5) \in \Psi(C_\Sigma)$
- A valuation satisfies a constraint: $\gamma \models \psi \in \Psi(C_\Sigma)$
- Example: $\gamma(x_a) = 3.2 \models x_a \leq 5$

Event-clock automata [AFH94]

Real-time automata, similar to Timed Automata [AD90], but

- Closed under all Boolean operations (e. g., complementation)
- Language inclusion is decidable, model checking possible
- Less expressive (e. g., no arbitrary clock resets)

Definition: Event-clock automaton $\mathcal{A}_{ec} = (\Sigma, Q, Q_0, E, F)$

- Σ, Q, Q_0, F as expected, and
- $E \subseteq Q \times \Sigma \times \Psi(C_\Sigma) \times 2^Q$ set of timed transitions.

Definition: Timed run

Given $w \in T\Sigma^\omega$, a timed run is of the form:

- $\theta : (q_0, \gamma_0) \xrightarrow{d_1} \xrightarrow{a_1} (q_1, \gamma_1) \xrightarrow{d_2} \xrightarrow{a_2} (q_2, \gamma_2) \xrightarrow{d_3} \xrightarrow{a_3} \dots$

γ_0 is initial, iff $\gamma_0(x_a) = \bot$, and $\gamma_0(y_a) = t_j$ (or $\gamma_0(y_a) = \bot$)

Event-clock automata [AFH94]

Real-time automata, similar to Timed Automata [AD90], but

- Closed under all Boolean operations (e. g., complementation)
- Language inclusion is decidable, model checking possible
- Less expressive (e. g., no arbitrary clock resets)

Definition: Event-clock automaton $\mathcal{A}_{ec} = (\Sigma, Q, Q_0, E, F)$

- Σ, Q, Q_0, F as expected, and
- $E \subseteq Q \times \Sigma \times \Psi(C_\Sigma) \times 2^Q$ set of timed transitions.

Definition: Timed run

Given $w \in T\Sigma^\omega$, a timed run is of the form:

- $\theta : (q_0, \gamma_0) \xrightarrow{d_1} \xrightarrow{a_1} (q_1, \gamma_1) \xrightarrow{d_2} \xrightarrow{a_2} (q_2, \gamma_2) \xrightarrow{d_3} \xrightarrow{a_3} \dots$

γ_0 is initial, iff $\gamma_0(x_a) = \perp$, and $\gamma_0(y_a) = t_j$ (or $\gamma_0(y_a) = \perp$)

Event-clock automata [AFH94]

Real-time automata, similar to Timed Automata [AD90], but

- Closed under all Boolean operations (e. g., complementation)
- Language inclusion is decidable, model checking possible
- Less expressive (e. g., no arbitrary clock resets)

Definition: Event-clock automaton $\mathcal{A}_{ec} = (\Sigma, Q, Q_0, E, F)$

- Σ, Q, Q_0, F as expected, and
- $E \subseteq Q \times \Sigma \times \Psi(C_\Sigma) \times 2^Q$ set of timed transitions.

Definition: Timed run

Given $w \in T\Sigma^\omega$, a timed run is of the form:

- $\theta : (q_0, \gamma_0) \xrightarrow{d_1} \xrightarrow{a_1} (q_1, \gamma_1) \xrightarrow{d_2} \xrightarrow{a_2} (q_2, \gamma_2) \xrightarrow{d_3} \xrightarrow{a_3} \dots$

γ_0 is initial, iff $\gamma_0(x_a) = \bot$, and $\gamma_0(y_a) = t_j$ (or $\gamma_0(y_a) = \bot$)

Event-clock automata [AFH94]

Real-time automata, similar to Timed Automata [AD90], but

- Closed under all Boolean operations (e. g., complementation)
- Language inclusion is decidable, model checking possible
- Less expressive (e. g., no arbitrary clock resets)

Definition: Event-clock automaton $\mathcal{A}_{ec} = (\Sigma, Q, Q_0, E, F)$

- Σ, Q, Q_0, F as expected, and
- $E \subseteq Q \times \Sigma \times \Psi(C_\Sigma) \times 2^Q$ set of timed transitions.

Definition: Timed run

Given $w \in T\Sigma^\omega$, a timed run is of the form:

- $\theta : (q_0, \gamma_0) \xrightarrow{d_1} \xrightarrow{a_1} (q_1, \gamma_1) \xrightarrow{d_2} \xrightarrow{a_2} (q_2, \gamma_2) \xrightarrow{d_3} \xrightarrow{a_3} \ldots$

γ_0 is initial, iff $\gamma_0(x_a) = \bot$, and $\gamma_0(y_a) = t_j$ (or $\gamma_0(y_a) = \bot$)

Event-clock automata [AFH94]

Real-time automata, similar to Timed Automata [AD90], but

- Closed under all Boolean operations (e. g., complementation)
- Language inclusion is decidable, model checking possible
- Less expressive (e. g., no arbitrary clock resets)

Definition: Event-clock automaton $\mathcal{A}_{ec} = (\Sigma, Q, Q_0, E, F)$

- Σ, Q, Q_0, F as expected, and
- $E \subseteq Q \times \Sigma \times \Psi(C_\Sigma) \times 2^Q$ set of timed transitions.

Definition: Timed run

Given $w \in T\Sigma^\omega$, a timed run is of the form:

- $\theta : (q_0, \gamma_0) \overset{d_1}{\rightarrow}\overset{a_1}{\rightarrow} (q_1, \gamma_1) \overset{d_2}{\rightarrow}\overset{a_2}{\rightarrow} (q_2, \gamma_2) \overset{d_3}{\rightarrow}\overset{a_3}{\rightarrow} \ldots$

γ_0 is initial, iff $\gamma_0(x_a) = \bot$, and $\gamma_0(y_a) = t_j$ (or $\gamma_0(y_a) = \bot$)

Event-clock automata [AFH94]

Real-time automata, similar to Timed Automata [AD90], but

- Closed under all Boolean operations (e. g., complementation)
- Language inclusion is decidable, model checking possible
- Less expressive (e. g., no arbitrary clock resets)

Definition: Event-clock automaton $\mathcal{A}_{ec} = (\Sigma, Q, Q_0, E, F)$

- Σ, Q, Q_0, F as expected, and
- $E \subseteq Q \times \Sigma \times \Psi(C_\Sigma) \times 2^Q$ set of timed transitions.

Definition: Timed run

Given $w \in T\Sigma^\omega$, a timed run is of the form:

- $\theta : (q_0, \gamma_0) \xrightarrow{d_1} \xrightarrow{a_1} (q_1, \gamma_1) \xrightarrow{d_2} \xrightarrow{a_2} (q_2, \gamma_2) \xrightarrow{d_3} \xrightarrow{a_3} \ldots$

γ_0 is initial, iff $\gamma_0(x_a) = \bot$, and $\gamma_0(y_a) = t_j$ (or $\gamma_0(y_a) = \bot$)

Event-clock automata [AFH94]

Real-time automata, similar to Timed Automata [AD90], but

- Closed under all Boolean operations (e. g., complementation)
- Language inclusion is decidable, model checking possible
- Less expressive (e. g., no arbitrary clock resets)

Definition: Event-clock automaton $\mathcal{A}_{ec} = (\Sigma, Q, Q_0, E, F)$

- Σ, Q, Q_0, F as expected, and
- $E \subseteq Q \times \Sigma \times \Psi(C_\Sigma) \times 2^Q$ set of timed transitions.

Definition: Timed run

Given $w \in T\Sigma^\omega$, a timed run is of the form:

- $\theta : (q_0, \gamma_0) \xrightarrow{d_1} \xrightarrow{a_1} (q_1, \gamma_1) \xrightarrow{d_2} \xrightarrow{a_2} (q_2, \gamma_2) \xrightarrow{d_3} \xrightarrow{a_3} \ldots$

γ_0 is initial, iff $\gamma_0(x_a) = \bot$, and $\gamma_0(y_a) = t_j$ (or $\gamma_0(y_a) = \bot$)

Event-clock automata [AFH94]

Real-time automata, similar to Timed Automata [AD90], but

- Closed under all Boolean operations (e. g., complementation)
- Language inclusion is decidable, model checking possible
- Less expressive (e. g., no arbitrary clock resets)

Definition: Event-clock automaton $\mathcal{A}_{ec} = (\Sigma, Q, Q_0, E, F)$

- Σ, Q, Q_0, F as expected, and
- $E \subseteq Q \times \Sigma \times \Psi(C_\Sigma) \times 2^Q$ set of timed transitions.

Definition: Timed run

Given $w \in T\Sigma^\omega$, a timed run is of the form:

- $\theta : (q_0, \gamma_0) \xrightarrow{d_1} \xrightarrow{a_1} (q_1, \gamma_1) \xrightarrow{d_2} \xrightarrow{a_2} (q_2, \gamma_2) \xrightarrow{d_3} \xrightarrow{a_3} \ldots$

γ_0 is initial, iff $\gamma_0(x_a) = \bot$, and $\gamma_0(y_a) = t_j$ (or $\gamma_0(y_a) = \bot$)

Event-clock automata [AFH94]

Real-time automata, similar to Timed Automata [AD90], but

- Closed under all Boolean operations (e. g., complementation)
- Language inclusion is decidable, model checking possible
- Less expressive (e. g., no arbitrary clock resets)

Definition: Event-clock automaton $\mathcal{A}_{ec} = (\Sigma, Q, Q_0, E, F)$

- Σ, Q, Q_0, F as expected, and
- $E \subseteq Q \times \Sigma \times \Psi(C_\Sigma) \times 2^Q$ set of timed transitions.

Definition: Timed run

Given $w \in T\Sigma^\omega$, a timed run is of the form:

- $\theta : (q_0, \gamma_0) \xrightarrow{d_1} \xrightarrow{a_1} (q_1, \gamma_1) \xrightarrow{d_2} \xrightarrow{a_2} (q_2, \gamma_2) \xrightarrow{d_3} \xrightarrow{a_3} \ldots$

γ_0 is initial, iff $\gamma_0(x_a) = \bot$, and $\gamma_0(y_a) = t_j$ (or $\gamma_0(y_a) = \bot$)

Timed LTL

Syntax: TLTL (aka state-clock logic [RS97])

$\varphi ::= a \mid \lhd_a \in [(l, r)] \mid \rhd_a \in [(l, r)] \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \mathbf{U} \varphi \mid \mathbf{X}\varphi, a \in \Sigma$

Semantics—intuitive account

Same as LTL, except for two real-time operators

- $\mathbf{G}(\rhd_a \in [0, 5])$: "always a within 5s"
- $\mathbf{G}((\lhd_q \in [0, 3]) \Rightarrow p)$: "always if q was within 3s, then p now"

Acceptors for TLTL

[R99]: $\varphi \mapsto \mathcal{A}_{ec}^{\varphi}$, s. t. $\mathcal{L}(\mathcal{A}_{ec}^{\varphi}) = \mathcal{L}(\varphi)$

Timed LTL

Syntax: TLTL (aka state-clock logic [RS97])

$\varphi ::= a \mid \lhd_a \in [(l,r)] \mid \rhd_a \in [(l,r)] \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \mathbf{U} \varphi \mid \mathbf{X}\varphi, a \in \Sigma$

Semantics—intuitive account

Same as LTL, except for two real-time operators

- $\mathbf{G}(\rhd_a \in [0,5])$: "always *a* within 5s"
- $\mathbf{G}((\lhd_q \in [0,3]) \Rightarrow p)$: "always if *q* was within 3s, then *p* now"

Acceptors for TLTL

[R99]: $\varphi \mapsto \mathcal{A}^{\varphi}_{ec}$, s. t. $\mathcal{L}(\mathcal{A}^{\varphi}_{ec}) = \mathcal{L}(\varphi)$

Timed LTL

Syntax: TLTL (aka state-clock logic [RS97])

$\varphi ::= a \mid \triangleleft_a \in [(l, r)] \mid \triangleright_a \in [(l, r)] \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \mathbf{U} \varphi \mid \mathbf{X}\varphi, a \in \Sigma$

Semantics—intuitive account

Same as LTL, except for two real-time operators

- $\mathbf{G}(\triangleright_a \in [0, 5])$: "always a within 5s"
- $\mathbf{G}((\triangleleft_q \in [0, 3]) \Rightarrow p)$: "always if q was within 3s, then p now"

Acceptors for TLTL

[R99]: $\varphi \mapsto \mathcal{A}_{ec}^{\varphi}$, s. t. $\mathcal{L}(\mathcal{A}_{ec}^{\varphi}) = \mathcal{L}(\varphi)$

Timed LTL

Syntax: TLTL (aka state-clock logic [RS97])

$\varphi ::= a \mid \lhd_a \in [(l, r)] \mid \rhd_a \in [(l, r)] \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \mathbf{U} \varphi \mid \mathbf{X}\varphi, a \in \Sigma$

Semantics—intuitive account

Same as LTL, except for two real-time operators

- $\mathbf{G}(\rhd_a \in [0, 5])$: "always a within 5s"
- $\mathbf{G}((\lhd_q \in [0, 3]) \Rightarrow p)$: "always if q was within 3s, then p now"

Acceptors for TLTL

[R99]: $\varphi \mapsto \mathcal{A}_{ec}^{\varphi}$, s. t. $\mathcal{L}(\mathcal{A}_{ec}^{\varphi}) = \mathcal{L}(\varphi)$

Timed LTL

Syntax: TLTL (aka state-clock logic [RS97])

$\varphi ::= a \mid \lhd_a \in [(l, r)] \mid \rhd_a \in [(l, r)] \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \mathbf{U} \varphi \mid \mathbf{X}\varphi,\ a \in \Sigma$

Semantics—intuitive account

Same as LTL, except for two real-time operators

- $\mathbf{G}(\rhd_a \in [0, 5])$: "always a within 5s"
- $\mathbf{G}((\lhd_q \in [0, 3]) \Rightarrow p)$: "always if q was within 3s, then p now"

Acceptors for TLTL

[R99]: $\varphi \mapsto \mathcal{A}^{\varphi}_{ec}$, s. t. $\mathcal{L}(\mathcal{A}^{\varphi}_{ec}) = \mathcal{L}(\varphi)$

Timed LTL

Syntax: TLTL (aka state-clock logic [RS97])

$$\varphi ::= a \mid \lhd_a \in [(l, r)] \mid \rhd_a \in [(l, r)] \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \mathbf{U} \varphi \mid \mathbf{X}\varphi, a \in \Sigma$$

Semantics—intuitive account

Same as LTL, except for two real-time operators

- $\mathbf{G}(\rhd_a \in [0, 5])$: "always *a* within 5s"
- $\mathbf{G}((\lhd_q \in [0, 3]) \Rightarrow p)$: "always if *q* was within 3s, then *p* now"

Acceptors for TLTL

[R99]: $\varphi \mapsto \mathcal{A}^{\varphi}_{ec}$, s. t. $\mathcal{L}(\mathcal{A}^{\varphi}_{ec}) = \mathcal{L}(\varphi)$

Monitoring TLTL properties

Monitoring TLTL properties

The runtime verification problem for TLTL

Find an on-the-fly decision procedure for TLTL_3:

$$[u \models \varphi] := \begin{cases} \top & \text{if } \forall w \in T\Sigma^\omega : uw \models \varphi \\ \bot & \text{if } \forall w \in T\Sigma^\omega : uw \not\models \varphi, \\ ? & \text{otherwise} \end{cases}$$

where $u \in T\Sigma^*$ and $\varphi \in TLTL$.

Monitoring TLTL properties

The runtime verification problem for TLTL

Find an on-the-fly decision procedure for TLTL_3:

$$[u \models \varphi] := \begin{cases} \top & \text{if } \forall w \in T\Sigma^\omega : uw \models \varphi \\ \bot & \text{if } \forall w \in T\Sigma^\omega : uw \not\models \varphi, \\ ? & \text{otherwise} \end{cases}$$

where $u \in T\Sigma^*$ and $\varphi \in TLTL$.

Given $w \in T\Sigma^\omega$, a timed run is of the form:

- $\theta : (q_0, \gamma_0) \xrightarrow{d_1} \xrightarrow{a_1} (q_1, \gamma_1) \xrightarrow{d_2} \xrightarrow{a_2} (q_2, \gamma_2) \xrightarrow{d_3} \xrightarrow{a_3} \dots$

γ_0 is initial, iff $\gamma_0(x_a) = \bot$, and $\gamma_0(y_a) = t_j$ (or $\gamma_0(y_a) = \bot$)

Monitoring TLTL properties

The runtime verification problem for TLTL

Find an on-the-fly decision procedure for TLTL_3:

$$[u \models \varphi] := \begin{cases} \top & \text{if } \forall w \in T\Sigma^\omega : uw \models \varphi \\ \bot & \text{if } \forall w \in T\Sigma^\omega : uw \not\models \varphi, \\ ? & \text{otherwise} \end{cases}$$

where $u \in T\Sigma^*$ and $\varphi \in TLTL$.

Given $w \in T\Sigma^\omega$, a timed run is of the form:

- $\theta : (q_0, \gamma_0) \xrightarrow{d_1} \xrightarrow{a_1} (q_1, \gamma_1) \xrightarrow{d_2} \xrightarrow{a_2} (q_2, \gamma_2) \xrightarrow{d_3} \xrightarrow{a_3} \ldots$

γ_0 is initial, iff $\gamma_0(x_a) = \bot$, and $\gamma_0(y_a) = t_j$ (or $\gamma_0(y_a) = \bot$)

Problem #1: Given i, how can we determine $\gamma_i(y_a)$?

Symbolic valuations

Use symbolic valuation, $\Gamma : C_\Sigma \rightarrow T_\perp \cup I$, assigning to each

- recording (x_a) clock variable a positive real, or bottom, and to each
- predicting (y_a) clock variable an interval, constraining the legal values for y_a (rather than an absolute value)

Definition: Operations on $\Gamma(x_a), \Gamma(y_a) = [(l, r)]$

- Elapse of time $t \in \mathbb{R}^{\geq 0}$:
  $\Gamma'(x_a) = \Gamma(x_a) + t, \Gamma'(y_a) = [(l \dot{-} t, r - t)]$
- (Reset) $\Gamma \downarrow a$: $x_a = 0, \Gamma'(y_a) = [0, \infty), \Gamma'(z \neq a) = \Gamma(z \neq a)$
- (Conjunction) $\Gamma' = \Gamma \wedge (\psi \in \Psi(C_\Sigma))$:
  $\Gamma'(y_a) = \Gamma(y_a) \wedge \bigwedge\{y_a \bowtie c \subseteq \psi\}$

Symbolic valuations

Use symbolic valuation, $\Gamma : C_\Sigma \to T_\perp \cup I$, assigning to each

- recording (x_a) clock variable a positive real, or bottom, and to each
- predicting (y_a) clock variable an interval, constraining the legal values for y_a (rather than an absolute value)

Definition: Operations on $\Gamma(x_a), \Gamma(y_a) = [(l, r)]$

- Elapse of time $t \in \mathbb{R}^{\geq 0}$:
  $\Gamma'(x_a) = \Gamma(x_a) + t, \Gamma'(y_a) = [(l \dot{-} t, r - t)]$
- (Reset) $\Gamma \downarrow a$: $x_a = 0, \Gamma'(y_a) = [0, \infty), \Gamma'(z \neq a) = \Gamma(z \neq a)$
- (Conjunction) $\Gamma' = \Gamma \wedge (\psi \in \Psi(C_\Sigma))$:
  $\Gamma'(y_a) = \Gamma(y_a) \wedge \bigwedge\{y_a \bowtie c \subseteq \psi\}$

Symbolic valuations

Use symbolic valuation, $\Gamma : C_\Sigma \to T_\perp \cup I$, assigning to each

- recording (x_a) clock variable a positive real, or bottom, and to each
- predicting (y_a) clock variable an interval, constraining the legal values for y_a (rather than an absolute value)

Definition: Operations on $\Gamma(x_a), \Gamma(y_a) = [(l, r)]$

- Elapse of time $t \in \mathbb{R}^{\geq 0}$:
  $\Gamma'(x_a) = \Gamma(x_a) + t, \Gamma'(y_a) = [(l \dot{-} t, r - t)]$
- (Reset) $\Gamma \downarrow a$: $x_a = 0, \Gamma'(y_a) = [0, \infty), \Gamma'(z \neq a) = \Gamma(z \neq a)$
- (Conjunction) $\Gamma' = \Gamma \wedge (\psi \in \Psi(C_\Sigma))$:
  $\Gamma'(y_a) = \Gamma(y_a) \wedge \bigwedge\{y_a \bowtie c \subseteq \psi\}$

Symbolic valuations

Use symbolic valuation, $\Gamma : C_\Sigma \rightarrow T_\perp \cup I$, assigning to each

- recording (x_a) clock variable a positive real, or bottom, and to each
- predicting (y_a) clock variable an interval, constraining the legal values for y_a (rather than an absolute value)

Definition: Operations on $\Gamma(x_a), \Gamma(y_a) = [(l, r)]$

- Elapse of time $t \in \mathbb{R}^{\geq 0}$:
  $\Gamma'(x_a) = \Gamma(x_a) + t, \Gamma'(y_a) = [(l \dot{-} t, r - t)]$
- (Reset) $\Gamma \downarrow a$: $x_a = 0, \Gamma'(y_a) = [0, \infty), \Gamma'(z \neq a) = \Gamma(z \neq a)$
- (Conjunction) $\Gamma' = \Gamma \wedge (\psi \in \Psi(C_\Sigma))$:
  $\Gamma'(y_a) = \Gamma(y_a) \wedge \bigwedge\{y_a \bowtie c \subseteq \psi\}$

Symbolic valuations

Use symbolic valuation, $\Gamma : C_\Sigma \to T_\perp \cup I$, assigning to each

- recording (x_a) clock variable a positive real, or bottom, and to each
- predicting (y_a) clock variable an interval, constraining the legal values for y_a (rather than an absolute value)

Definition: Operations on $\Gamma(x_a), \Gamma(y_a) = [(l, r)]$

- Elapse of time $t \in \mathbb{R}^{\geq 0}$:
  $\Gamma'(x_a) = \Gamma(x_a) + t, \Gamma'(y_a) = [(l \dot{-} t, r - t)]$
- (Reset) $\Gamma \downarrow a$: $x_a = 0, \Gamma'(y_a) = [0, \infty), \Gamma'(z \neq a) = \Gamma(z \neq a)$
- (Conjunction) $\Gamma' = \Gamma \wedge (\psi \in \Psi(C_\Sigma))$:
  $\Gamma'(y_a) = \Gamma(y_a) \wedge \bigwedge\{y_a \bowtie c \subseteq \psi\}$

Symbolic valuations

Use symbolic valuation, $\Gamma : C_\Sigma \to T_\perp \cup I$, assigning to each

- recording (x_a) clock variable a positive real, or bottom, and to each
- predicting (y_a) clock variable an interval, constraining the legal values for y_a (rather than an absolute value)

Definition: Operations on $\Gamma(x_a), \Gamma(y_a) = [(l, r)]$

- Elapse of time $t \in \mathbb{R}^{\geq 0}$:
  $\Gamma'(x_a) = \Gamma(x_a) + t, \Gamma'(y_a) = [(l \dot{-} t, r - t)]$
- (Reset) $\Gamma \downarrow a$: $x_a = 0, \Gamma'(y_a) = [0, \infty), \Gamma'(z \neq a) = \Gamma(z \neq a)$
- (Conjunction) $\Gamma' = \Gamma \wedge (\psi \in \Psi(C_\Sigma))$:
  $\Gamma'(y_a) = \Gamma(y_a) \wedge \bigwedge \{y_a \bowtie c \subseteq \psi\}$

Symbolic valuations

Use symbolic valuation, $\Gamma : C_\Sigma \to T_\perp \cup I$, assigning to each

- recording (x_a) clock variable a positive real, or bottom, and to each
- predicting (y_a) clock variable an interval, constraining the legal values for y_a (rather than an absolute value)

Definition: Operations on $\Gamma(x_a), \Gamma(y_a) = [(l, r)]$

- Elapse of time $t \in \mathbb{R}^{\geq 0}$:
  $\Gamma'(x_a) = \Gamma(x_a) + t, \Gamma'(y_a) = [(l \dot{-} t, r - t)]$
- (Reset) $\Gamma \downarrow a$: $x_a = 0, \Gamma'(y_a) = [0, \infty), \Gamma'(z \neq a) = \Gamma(z \neq a)$
- (Conjunction) $\Gamma' = \Gamma \wedge (\psi \in \Psi(C_\Sigma))$:
  $\Gamma'(y_a) = \Gamma(y_a) \wedge \bigwedge \{y_a \bowtie c \subseteq \psi\}$

Symbolic runs

Instead of state-valuation tuples, (q, γ), we use state-symbolic-valuation tuples:

$$(q_0, \Gamma_0) \xrightarrow{\alpha_1} (q_1, \Gamma_1) \xrightarrow{\alpha_2} (q_2, \Gamma_2) \xrightarrow{\alpha_3} \ldots \qquad \alpha_i = (a_i, t_i)$$

Let $\mathcal{A}_{ec} = (\Sigma, Q, Q_0, E, F)$ and $E \subseteq Q \times \Sigma \times \Psi(C_\Sigma) \times 2^Q$

- A transition $(q, a, \psi, \{q'\}) \in E$ is applicable to a pair (q, Γ) if $\Gamma \models x_b \bowtie c \in \psi \wedge 0 \in \Gamma(y_a)$
- Successor of (q, Γ) is (q', Γ') with $\Gamma' = (\Gamma \downarrow a) \wedge \psi$ (Reset + Conjunction)
- Γ_0 is initial, iff for all $a \in \Sigma$, $\Gamma_0(x_a) = \perp$, $\Gamma_0(y_a) = [0, \infty)$

Corollary

γ_0 is dependent on $w \in T\Sigma^\omega$, and Γ_0 is not.

Symbolic runs

Instead of state-valuation tuples, (q, γ), we use state-symbolic-valuation tuples:

$$(q_0, \Gamma_0) \xrightarrow{\alpha_1} (q_1, \Gamma_1) \xrightarrow{\alpha_2} (q_2, \Gamma_2) \xrightarrow{\alpha_3} \ldots \qquad \alpha_i = (a_i, t_i)$$

Let $\mathcal{A}_{ec} = (\Sigma, Q, Q_0, E, F)$ and $E \subseteq Q \times \Sigma \times \Psi(C_\Sigma) \times 2^Q$

- A transition $(q, a, \psi, \{q'\}) \in E$ is applicable to a pair (q, Γ) if $\Gamma \models x_b \bowtie c \in \psi \wedge 0 \in \Gamma(y_a)$
- Successor of (q, Γ) is (q', Γ') with $\Gamma' = (\Gamma \downarrow a) \wedge \psi$ (Reset + Conjunction)
- Γ_0 is initial, iff for all $a \in \Sigma$, $\Gamma_0(x_a) = \bot$, $\Gamma_0(y_a) = [0, \infty)$

Corollary

γ_0 is dependent on $w \in T\Sigma^\omega$, and Γ_0 is not.

Symbolic runs

Instead of state-valuation tuples, (q, γ), we use state-symbolic-valuation tuples:

$$(q_0, \Gamma_0) \xrightarrow{\alpha_1} (q_1, \Gamma_1) \xrightarrow{\alpha_2} (q_2, \Gamma_2) \xrightarrow{\alpha_3} \dots \qquad \alpha_i = (a_i, t_i)$$

Let $\mathcal{A}_{ec} = (\Sigma, Q, Q_0, E, F)$ and $E \subseteq Q \times \Sigma \times \Psi(C_\Sigma) \times 2^Q$

- A transition $(q, a, \psi, \{q'\}) \in E$ is applicable to a pair (q, Γ) if $\Gamma \models x_b \bowtie c \in \psi \wedge 0 \in \Gamma(y_a)$
- Successor of (q, Γ) is (q', Γ') with $\Gamma' = (\Gamma \downarrow a) \wedge \psi$ (Reset + Conjunction)
- Γ_0 is initial, iff for all $a \in \Sigma$, $\Gamma_0(x_a) = \bot$, $\Gamma_0(y_a) = [0, \infty)$

Corollary

γ_0 is dependent on $w \in T\Sigma^\omega$, and Γ_0 is not.

Symbolic runs

Instead of state-valuation tuples, (q, γ), we use state-symbolic-valuation tuples:

$$(q_0, \Gamma_0) \xrightarrow{\alpha_1} (q_1, \Gamma_1) \xrightarrow{\alpha_2} (q_2, \Gamma_2) \xrightarrow{\alpha_3} \dots \qquad \alpha_i = (a_i, t_i)$$

Let $\mathcal{A}_{ec} = (\Sigma, Q, Q_0, E, F)$ and $E \subseteq Q \times \Sigma \times \Psi(C_\Sigma) \times 2^Q$

- A transition $(q, a, \psi, \{q'\}) \in E$ is applicable to a pair (q, Γ) if $\Gamma \models x_b \bowtie c \in \psi \wedge 0 \in \Gamma(y_a)$
- Successor of (q, Γ) is (q', Γ') with $\Gamma' = (\Gamma \downarrow a) \wedge \psi$ (Reset + Conjunction)
- Γ_0 is initial, iff for all $a \in \Sigma$, $\Gamma_0(x_a) = \bot$, $\Gamma_0(y_a) = [0, \infty)$

Corollary

γ_0 is dependent on $w \in T\Sigma^\omega$, and Γ_0 is not.

Symbolic runs

Instead of state-valuation tuples, (q, γ), we use state-symbolic-valuation tuples:

$$(q_0, \Gamma_0) \xrightarrow{\alpha_1} (q_1, \Gamma_1) \xrightarrow{\alpha_2} (q_2, \Gamma_2) \xrightarrow{\alpha_3} \dots \qquad \alpha_i = (a_i, t_i)$$

Let $\mathcal{A}_{ec} = (\Sigma, Q, Q_0, E, F)$ and $E \subseteq Q \times \Sigma \times \Psi(C_\Sigma) \times 2^Q$

- A transition $(q, a, \psi, \{q'\}) \in E$ is applicable to a pair (q, Γ) if $\Gamma \models x_b \bowtie c \in \psi \wedge 0 \in \Gamma(y_a)$
- Successor of (q, Γ) is (q', Γ') with $\Gamma' = (\Gamma \downarrow a) \wedge \psi$
  (Reset + Conjunction)
- Γ_0 is initial, iff for all $a \in \Sigma$, $\Gamma_0(x_a) = \bot$, $\Gamma_0(y_a) = [0, \infty)$

Corollary

γ_0 is dependent on $w \in T\Sigma^\omega$, and Γ_0 is not.

Symbolic runs

Instead of state-valuation tuples, (q, γ), we use state-symbolic-valuation tuples:

$$(q_0, \Gamma_0) \xrightarrow{\alpha_1} (q_1, \Gamma_1) \xrightarrow{\alpha_2} (q_2, \Gamma_2) \xrightarrow{\alpha_3} \dots \qquad \alpha_i = (a_i, t_i)$$

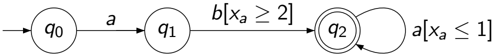
Let $\mathcal{A}_{ec} = (\Sigma, Q, Q_0, E, F)$ and $E \subseteq Q \times \Sigma \times \Psi(C_\Sigma) \times 2^Q$

- A transition $(q, a, \psi, \{q'\}) \in E$ is applicable to a pair (q, Γ) if $\Gamma \models x_b \bowtie c \in \psi \wedge 0 \in \Gamma(y_a)$
- Successor of (q, Γ) is (q', Γ') with $\Gamma' = (\Gamma \downarrow a) \wedge \psi$ (Reset + Conjunction)
- Γ_0 is initial, iff for all $a \in \Sigma$, $\Gamma_0(x_a) = \bot$, $\Gamma_0(y_a) = [0, \infty)$

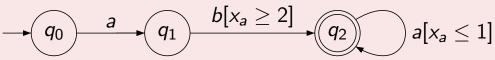
Corollary

γ_0 is dependent on $w \in T\Sigma^\omega$, and Γ_0 is not.

Checking emptiness per state

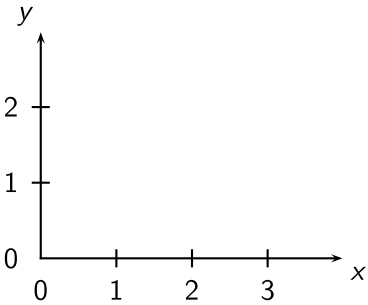


Checking emptiness per state



Problem #2: Although the language of $\mathcal{A}_{ec}(q_2)$ is non-empty, there does not exist an accepting run.

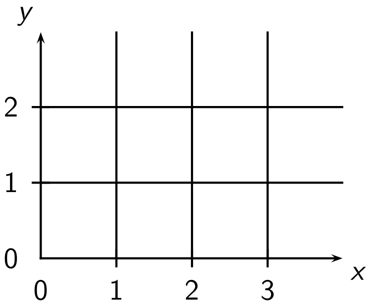
Region equivalence [AD94]



Build equivalence relation which is of finite index and is

- "compatible" with clock constraints:
  $r, r' \in R \Rightarrow \forall$ constraints $\gamma : r \models \gamma \Leftrightarrow r' \models \gamma$
- compatible with time elapsing:
  $r, r' \in R \Rightarrow$ same delay successor region

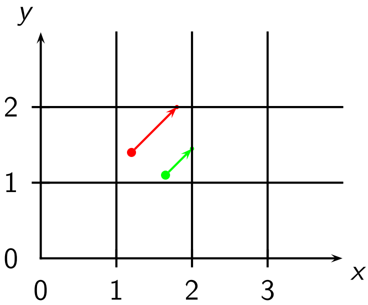
Region equivalence [AD94]



Build equivalence relation which is of finite index and is

- "compatible" with clock constraints:
  $r, r' \in R \Rightarrow \forall$ constraints $\gamma : r \models \gamma \Leftrightarrow r' \models \gamma$
- compatible with time elapsing:
  $r, r' \in R \Rightarrow$ same delay successor region

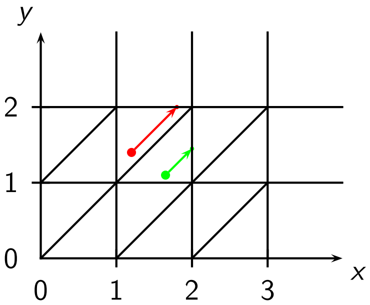
Region equivalence [AD94]



Build equivalence relation which is of finite index and is

- "compatible" with clock constraints:
  $r, r' \in R \Rightarrow \forall$ constraints $\gamma : r \models \gamma \Leftrightarrow r' \models \gamma$
- compatible with time elapsing:
  $r, r' \in R \Rightarrow$ same delay successor region

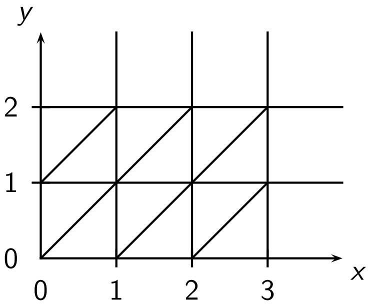
Region equivalence [AD94]



Build equivalence relation which is of finite index and is

- "compatible" with clock constraints:
  $r, r' \in R \Rightarrow \forall$ constraints $\gamma : r \models \gamma \Leftrightarrow r' \models \gamma$
- compatible with time elapsing:
  $r, r' \in R \Rightarrow$ same delay successor region

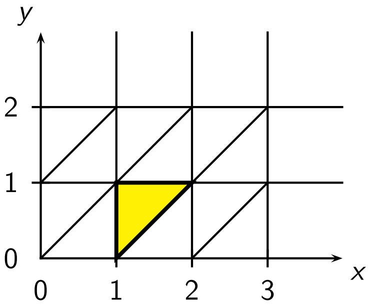
Region equivalence [AD94]



Build equivalence relation which is of finite index and is

- "compatible" with clock constraints:
  $r, r' \in R \Rightarrow \forall$ constraints $\gamma : r \models \gamma \Leftrightarrow r' \models \gamma$
- compatible with time elapsing:
  $r, r' \in R \Rightarrow$ same delay successor region

Region equivalence [AD94]

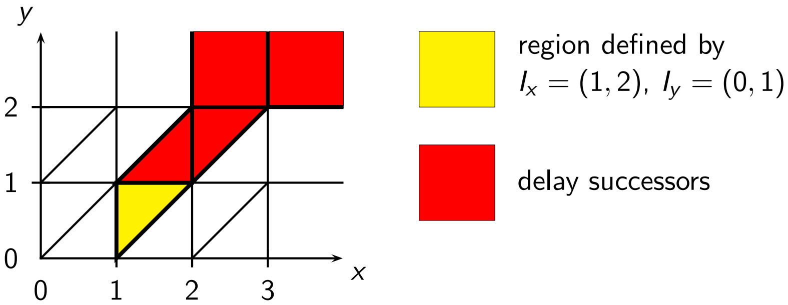


region defined by $I_x = (1,2),\ I_y = (0,1)$

Build equivalence relation which is of finite index and is

- "compatible" with clock constraints:
  $r, r' \in R \Rightarrow \forall$ constraints $\gamma : r \models \gamma \Leftrightarrow r' \models \gamma$
- compatible with time elapsing:
  $r, r' \in R \Rightarrow$ same delay successor region

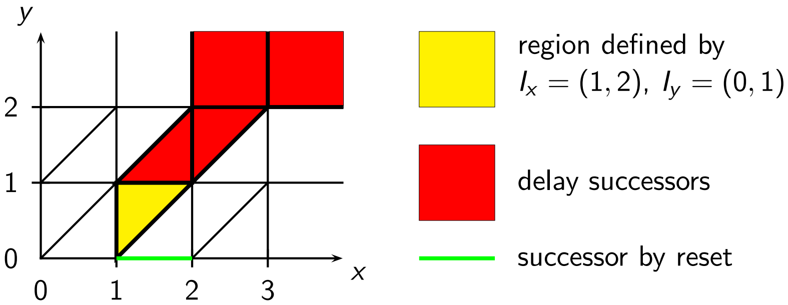
Region equivalence [AD94]



Build equivalence relation which is of finite index and is

- "compatible" with clock constraints:
  $r, r' \in R \Rightarrow \forall$ constraints $\gamma : r \models \gamma \Leftrightarrow r' \models \gamma$
- compatible with time elapsing:
  $r, r' \in R \Rightarrow$ same delay successor region

Region equivalence [AD94]



Build equivalence relation which is of finite index and is

- "compatible" with clock constraints:
  $r, r' \in R \Rightarrow \forall$ constraints $\gamma : r \models \gamma \Leftrightarrow r' \models \gamma$
- compatible with time elapsing:
  $r, r' \in R \Rightarrow$ same delay successor region

Region automaton

Construction: $\mathcal{A}_{ec} = (\Sigma, Q, Q_0, E, F) \mapsto RA$

- For each transition $(q, a, \psi, \{q'\}) \in E$
- Build transitions in the RA: $(q, R) \xrightarrow{a} (q', R')$ if
  - there exists R'' a delay successor of R s. t.
  - R'' satisfies the constraint ψ (i. e., $R'' \subseteq \psi$)
  - R'' (mod. reset + conjunction of clocks) is included in R'

Theorem

An ECA and its region automaton RA are time-abstract bisimilar

- $\mathcal{L}(RA^{\varphi}) = ut(\mathcal{L}(\mathcal{A}_{ec}^{\varphi}))$ $(w = (a, 1.2)(b, 3.4);\ ut(w) = ab)$
- The region automaton is finite
- Language emptiness can be decided on the RA

Region automaton

Construction: $\mathcal{A}_{ec} = (\Sigma, Q, Q_0, E, F) \mapsto RA$

- For each transition $(q, a, \psi, \{q'\}) \in E$
- Build transitions in the RA: $(q, R) \xrightarrow{a} (q', R')$ if
  - there exists R'' a delay successor of R s. t.
  - R'' satisfies the constraint ψ (i. e., $R'' \subseteq \psi$)
  - R'' (mod. reset + conjunction of clocks) is included in R'

Theorem

An ECA and its region automaton RA are time-abstract bisimilar

- $\mathcal{L}(RA^{\varphi}) = ut(\mathcal{L}(\mathcal{A}_{ec}^{\varphi}))$ ($w = (a, 1.2)(b, 3.4)$; $ut(w) = ab$)
- The region automaton is finite
- Language emptiness can be decided on the RA

Region automaton

Construction: $\mathcal{A}_{ec} = (\Sigma, Q, Q_0, E, F) \mapsto RA$

- For each transition $(q, a, \psi, \{q'\}) \in E$
- Build transitions in the RA: $(q, R) \xrightarrow{a} (q', R')$ if
  - there exists R'' a delay successor of R s. t.
  - R'' satisfies the constraint ψ (i. e., $R'' \subseteq \psi$)
  - R'' (mod. reset + conjunction of clocks) is included in R'

Theorem

An ECA and its region automaton RA are time-abstract bisimilar

- $\mathcal{L}(RA^{\varphi}) = ut(\mathcal{L}(\mathcal{A}_{ec}^{\varphi}))$ $(w = (a, 1.2)(b, 3.4);\ ut(w) = ab)$
- The region automaton is finite
- Language emptiness can be decided on the RA

Region automaton

Construction: $\mathcal{A}_{ec} = (\Sigma, Q, Q_0, E, F) \mapsto RA$

- For each transition $(q, a, \psi, \{q'\}) \in E$
- Build transitions in the RA: $(q, R) \xrightarrow{a} (q', R')$ if
  - there exists R'' a delay successor of R s. t.
  - R'' satisfies the constraint ψ (i. e., $R'' \subseteq \psi$)
  - R'' (mod. reset + conjunction of clocks) is included in R'

Theorem

An ECA and its region automaton RA are time-abstract bisimilar

- $\mathcal{L}(RA^{\varphi}) = ut(\mathcal{L}(\mathcal{A}_{ec}^{\varphi}))$ $(w = (a, 1.2)(b, 3.4);\ ut(w) = ab)$
- The region automaton is finite
- Language emptiness can be decided on the RA

Region automaton

Construction: $\mathcal{A}_{ec} = (\Sigma, Q, Q_0, E, F) \mapsto RA$

- For each transition $(q, a, \psi, \{q'\}) \in E$
- Build transitions in the RA: $(q, R) \xrightarrow{a} (q', R')$ if
  - there exists R'' a delay successor of R s. t.
  - R'' satisfies the constraint ψ (i. e., $R'' \subseteq \psi$)
  - R'' (mod. reset + conjunction of clocks) is included in R'

Theorem

An ECA and its region automaton RA are time-abstract bisimilar

- $\mathcal{L}(RA^{\varphi}) = ut(\mathcal{L}(\mathcal{A}_{ec}^{\varphi}))$ $(w = (a, 1.2)(b, 3.4);\ ut(w) = ab)$
- The region automaton is finite
- Language emptiness can be decided on the RA

Region automaton

Construction: $\mathcal{A}_{ec} = (\Sigma, Q, Q_0, E, F) \mapsto RA$

- For each transition $(q, a, \psi, \{q'\}) \in E$
- Build transitions in the RA: $(q, R) \xrightarrow{a} (q', R')$ if
  - there exists R'' a delay successor of R s. t.
  - R'' satisfies the constraint ψ (i. e., $R'' \subseteq \psi$)
  - R'' (mod. reset + conjunction of clocks) is included in R'

Theorem

An ECA and its region automaton RA are time-abstract bisimilar

- $\mathcal{L}(RA^{\varphi}) = ut(\mathcal{L}(\mathcal{A}_{ec}^{\varphi}))$ $(w = (a, 1.2)(b, 3.4);\ ut(w) = ab)$
- The region automaton is finite
- Language emptiness can be decided on the RA

Region automaton

Construction: $\mathcal{A}_{ec} = (\Sigma, Q, Q_0, E, F) \mapsto RA$

- For each transition $(q, a, \psi, \{q'\}) \in E$
- Build transitions in the RA: $(q, R) \xrightarrow{a} (q', R')$ if
  - there exists R'' a delay successor of R s. t.
  - R'' satisfies the constraint ψ (i. e., $R'' \subseteq \psi$)
  - R'' (mod. reset + conjunction of clocks) is included in R'

Theorem

An ECA and its region automaton RA are time-abstract bisimilar

- $\mathcal{L}(RA^{\varphi}) = ut(\mathcal{L}(\mathcal{A}_{ec}^{\varphi}))$ ($w = (a, 1.2)(b, 3.4)$; $ut(w) = ab$)
- The region automaton is finite
- Language emptiness can be decided on the RA

Region automaton

Construction: $\mathcal{A}_{ec} = (\Sigma, Q, Q_0, E, F) \mapsto RA$

- For each transition $(q, a, \psi, \{q'\}) \in E$
- Build transitions in the RA: $(q, R) \xrightarrow{a} (q', R')$ if
  - there exists R'' a delay successor of R s. t.
  - R'' satisfies the constraint ψ (i. e., $R'' \subseteq \psi$)
  - R'' (mod. reset + conjunction of clocks) is included in R'

Theorem

An ECA and its region automaton RA are time-abstract bisimilar

- $\mathcal{L}(RA^{\varphi}) = ut(\mathcal{L}(\mathcal{A}_{ec}^{\varphi}))$ $(w = (a, 1.2)(b, 3.4);\ ut(w) = ab)$
- The region automaton is finite
- Language emptiness can be decided on the RA

Region automaton

Construction: $\mathcal{A}_{ec} = (\Sigma, Q, Q_0, E, F) \mapsto RA$

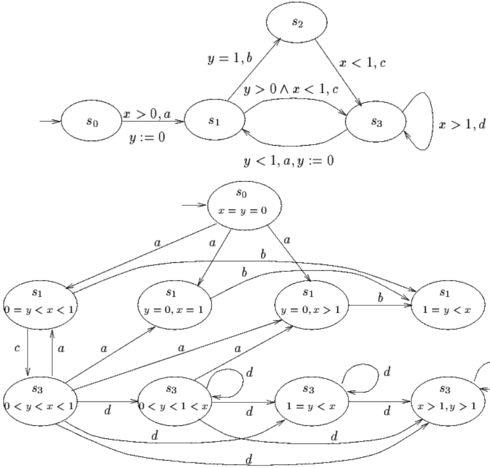
- For each transition $(q, a, \psi, \{q'\}) \in E$
- Build transitions in the RA: $(q, R) \xrightarrow{a} (q', R')$ if
  - there exists R'' a delay successor of R s. t.
  - R'' satisfies the constraint ψ (i. e., $R'' \subseteq \psi$)
  - R'' (mod. reset + conjunction of clocks) is included in R'

Theorem

An ECA and its region automaton RA are time-abstract bisimilar

- $\mathcal{L}(RA^{\varphi}) = ut(\mathcal{L}(\mathcal{A}_{ec}^{\varphi}))$ $(w = (a, 1.2)(b, 3.4);\ ut(w) = ab)$
- The region automaton is finite
- Language emptiness can be decided on the RA

Region automaton example [A99]



Monitoring TLTL—putting it all together

- Monitoring is based on $\mathcal{A}_{ec}^{\varphi}$ and RA^{φ}
- No explicit monitor construction

Algorithm: Automata execution

Let Γ_0 be initial symbolic valuation of $\mathcal{A}_{ec}^{\varphi}$, and l_0 an initial state of $\mathcal{A}_{ec}^{\varphi}$.

A1. [Compute successor set.] For the first event (a_0, t_0), the set of successors w. r. t. $\mathcal{A}_{ec}^{\varphi}$ is computed.

A2. [Set empty?] If set is empty, the underlying formula is violated, and *false* issued. If not, go to step A3.

A3. [Check emptiness.] Each successor is a pair (l, Γ) and corresponds to a set of states in RA^{φ}. Iff for all of them the accepted language is empty, the underlying property is violated, and *false* issued.

A4. [Process next event.] Issue *true*, and continue procedure from A2 with each successor state (l, Γ) for which a corresponding accepting state of RA^{φ} exists, reading a new input event.

Monitoring TLTL—putting it all together

- Monitoring is based on $\mathcal{A}_{ec}^{\varphi}$ and RA^{φ}
- No explicit monitor construction

Algorithm: Automata execution

Let Γ_0 be initial symbolic valuation of $\mathcal{A}_{ec}^{\varphi}$, and l_0 an initial state of $\mathcal{A}_{ec}^{\varphi}$.

A1. [Compute successor set.] For the first event (a_0, t_0), the set of successors w. r. t. $\mathcal{A}_{ec}^{\varphi}$ is computed.

A2. [Set empty?] If set is empty, the underlying formula is violated, and *false* issued. If not, go to step A3.

A3. [Check emptiness.] Each successor is a pair (l, Γ) and corresponds to a set of states in RA^{φ}. Iff for all of them the accepted language is empty, the underlying property is violated, and *false* issued.

A4. [Process next event.] Issue *true*, and continue procedure from A2 with each successor state (l, Γ) for which a corresponding accepting state of RA^{φ} exists, reading a new input event.

Monitoring TLTL—putting it all together

- Monitoring is based on $\mathcal{A}_{ec}^{\varphi}$ and RA^{φ}
- No explicit monitor construction

Algorithm: Automata execution

Let Γ_0 be initial symbolic valuation of $\mathcal{A}_{ec}^{\varphi}$, and l_0 an initial state of $\mathcal{A}_{ec}^{\varphi}$.

A1. [Compute successor set.] For the first event (a_0, t_0), the set of successors w. r. t. $\mathcal{A}_{ec}^{\varphi}$ is computed.

A2. [Set empty?] If set is empty, the underlying formula is violated, and *false* issued. If not, go to step A3.

A3. [Check emptiness.] Each successor is a pair (l, Γ) and corresponds to a set of states in RA^{φ}. Iff for all of them the accepted language is empty, the underlying property is violated, and *false* issued.

A4. [Process next event.] Issue *true*, and continue procedure from A2 with each successor state (l, Γ) for which a corresponding accepting state of RA^{φ} exists, reading a new input event.

Monitoring TLTL—putting it all together

- Monitoring is based on $\mathcal{A}^{\varphi}_{ec}$ and RA^{φ}
- No explicit monitor construction

Algorithm: Automata execution

Let Γ_0 be initial symbolic valuation of $\mathcal{A}^{\varphi}_{ec}$, and l_0 an initial state of $\mathcal{A}^{\varphi}_{ec}$.

A1. [Compute successor set.] For the first event (a_0, t_0), the set of successors w. r. t. $\mathcal{A}^{\varphi}_{ec}$ is computed.

A2. [Set empty?] If set is empty, the underlying formula is violated, and *false* issued. If not, go to step A3.

A3. [Check emptiness.] Each successor is a pair (l, Γ) and corresponds to a set of states in RA^{φ}. Iff for all of them the accepted language is empty, the underlying property is violated, and *false* issued.

A4. [Process next event.] Issue *true*, and continue procedure from A2 with each successor state (l, Γ) for which a corresponding accepting state of RA^{φ} exists, reading a new input event.

Monitoring TLTL—putting it all together

- Monitoring is based on $\mathcal{A}_{ec}^{\varphi}$ and RA^{φ}
- No explicit monitor construction

Algorithm: Automata execution

Let Γ_0 be initial symbolic valuation of $\mathcal{A}_{ec}^{\varphi}$, and l_0 an initial state of $\mathcal{A}_{ec}^{\varphi}$.

A1. [Compute successor set.] For the first event (a_0, t_0), the set of successors w. r. t. $\mathcal{A}_{ec}^{\varphi}$ is computed.

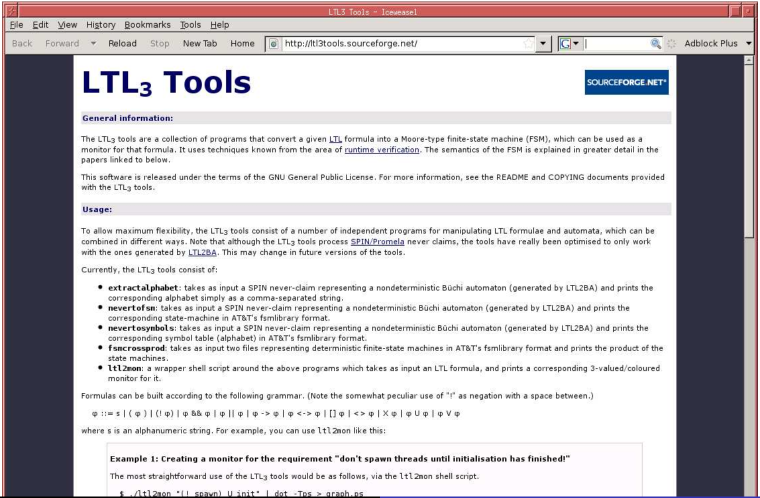
A2. [Set empty?] If set is empty, the underlying formula is violated, and *false* issued. If not, go to step A3.

A3. [Check emptiness.] Each successor is a pair (l, Γ) and corresponds to a set of states in RA^{φ}. Iff for all of them the accepted language is empty, the underlying property is violated, and *false* issued.

A4. [Process next event.] Issue *true*, and continue procedure from A2 with each successor state (l, Γ) for which a corresponding accepting state of RA^{φ} exists, reading a new input event.

Many thanks!

Try it out: `http://ltl3tools.sf.net/`!



LTL$_3$ Tools

General information:

The LTL$_3$ tools are a collection of programs that convert a given LTL formula into a Moore-type finite-state machine (FSM), which can be used as a monitor for that formula. It uses techniques known from the area of runtime verification. The semantics of the FSM is explained in greater detail in the papers linked to below.

This software is released under the terms of the GNU General Public License. For more information, see the README and COPYING documents provided with the LTL$_3$ tools.

Usage:

To allow maximum flexibility, the LTL$_3$ tools consist of a number of independent programs for manipulating LTL formulae and automata, which can be combined in different ways. Note that although the LTL$_3$ tools process SPIN/Promela never claims, the tools have really been optimised to only work with the ones generated by LTL2BA. This may change in future versions of the tools.

Currently, the LTL$_3$ tools consist of:

- **extractalphabet**: takes as input a SPIN never-claim representing a nondeterministic Büchi automaton (generated by LTL2BA) and prints the corresponding alphabet simply as a comma-separated string.
- **nevertofsm**: takes as input a SPIN never-claim representing a nondeterministic Büchi automaton (generated by LTL2BA) and prints the corresponding state-machine in AT&T's fsmlibrary format.
- **nevertosymbols**: takes as input a SPIN never-claim representing a nondeterministic Büchi automaton (generated by LTL2BA) and prints the corresponding symbol table (alphabet) in AT&T's fsmlibrary format.
- **fsmcrossprod**: takes as input two files representing deterministic finite-state machines in AT&T's fsmlibrary format and prints the product of the state machines.
- **ltl2mon**: a wrapper shell script around the above programs which takes as input an LTL formula, and prints a corresponding 3-valued/coloured monitor for it.

Formulas can be built according to the following grammar. (Note the somewhat peculiar use of "!" as negation with a space between.)

φ ::= s | (φ) | (! φ) | φ && φ | φ || φ | φ -> φ | φ <-> φ | [] φ | <> φ | X φ | φ U φ | φ V φ

where s is an alphanumeric string. For example, you can use `ltl2mon` like this:

Example 1: Creating a monitor for the requirement "don't spawn threads until initialisation has finished!"

The most straightforward use of the LTL$_3$ tools would be as follows, via the `ltl2mon` shell script.

```
$ ./ltl2mon "(! spawn) U init" | dot -Tps > graph.ps
```