Abstraction for system verification

Susanne Graf

VERIMAG, CNRS

Marktoberdorf, August 2010

Outline

1. Motivation
2. Property preserving abstractions: semantic level
   - Galois connexions between lattices
   - Abstractions for transition systems
3. Effectively computing abstractions
4. Verification of composed systems

1. Motivation

2. Property preserving abstractions: semantic level
   - Galois connexions between lattices
   - Abstractions for transition systems

3. Effectively computing abstractions

4. Verification of composed systems

What is verification?

We consider a specification / verification setting to be given by:

(1) a set of potential *design specifications*, called "models" $\mathcal{M}$, with $M \in \mathcal{M}$ (how)

(2) a set of potential *requirements* $\mathcal{L}$, with $\varphi \in \mathcal{L}$ (what)

(3*) a *satisfaction* or *conformance* relation $\models \subseteq 2^{\mathcal{M} \times \mathcal{L}}$ relating models and properties. We write $M \models \varphi$ and $M \not\models \varphi$.

(4) an *algorithm* to check $M \models \varphi$ (*model-checking*)

* Sometimes, we also consider the cases

- $M \models M'$ (refinement)
- $\varphi \models \varphi'$ (requirements engineering)

What is verification?

We consider a specification / verification setting to be given by:

(1) a set of potential *design specifications*, called "models" $\mathcal{M}$, with $M \in \mathcal{M}$ (how)

(2) a set of potential *requirements* $\mathcal{L}$, with $\varphi \in \mathcal{L}$ (what)

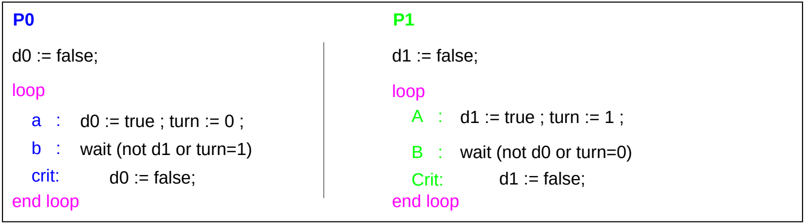
(3*) a *satisfaction* or *conformance* relation $\models \subseteq 2^{\mathcal{M} \times \mathcal{L}}$ relating models and properties. We write $M \models \varphi$ and $M \not\models \varphi$.

(4) an *algorithm* to check $M \models \varphi$ (*model-checking*)

* Sometimes, we also consider the cases

- $M \models M'$ (refinement)
- $\varphi \models \varphi'$ (requirements engineering)

Example for M and φ: Peterson mutex algorithm



P0

```
d0 := false;
loop
   a   :   d0 := true ; turn := 0 ;
   b   :   wait (not d1 or turn=1)
   crit:        d0 := false;
end loop
```

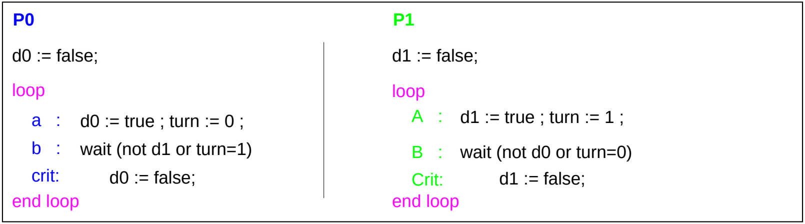
P1

```
d1 := false;
loop
   A   :   d1 := true ; turn := 1 ;
   B   :   wait (not d0 or turn=0)
   Crit:        d1 := false;
end loop
```

Does the *design* M guarantee the following *requirements*?

- *mutual exclusion*: at most one process is in critical section *crit*
- *deadlock freedom*: system is never definitively blocked
- *non-blocking*: always, any of the critical sections is *reachable* in a bounded number of steps
- *fairness*: when a process is engaged (in *b* resp. *B*) its critical section *must be reached* in a bounded number of steps.

Example for M and φ: Peterson mutex algorithm



```
P0                                          P1

d0 := false;                                d1 := false;

loop                                        loop
   a   :   d0 := true ; turn := 0 ;            A   :   d1 := true ; turn := 1 ;
   b   :   wait (not d1 or turn=1)             B   :   wait (not d0 or turn=0)
   crit:       d0 := false;                    Crit:       d1 := false;
end loop                                    end loop
```

Does the *design* M guarantee the following *requirements*?

- *mutual exclusion*: at most one process is in critical section *crit*
- *deadlock freedom*: system is never definitively blocked
- *non-blocking*: always, any of the critical sections is *reachable* in a bounded number of steps
- *fairness*: when a process is engaged (in *b* resp. *B*) its critical section *must be reached* in a bounded number of steps.

(1) Design specifications

... express:

- a (set of) potential solution(s) (use a shared variable “turn” ....)
- i.e. specific algorithms/ components/ ...
- can (in principle) be “implemented”

Typical formalisms:

- programs, abstract programs
- (extended) automata, transition systems (TS), Kripke structures, Petri Nets, ...
- Composition: parallel composition ($P_0 \| P_1$)

Design specifications are often operational.

(1) Design specifications

... express:

- a (set of) potential solution(s) (use a shared variable “turn” ....)
- i.e. specific algorithms/ components/ ...
- can (in principle) be “implemented”

Typical formalisms:

- programs, abstract programs
- (extended) automata, transition systems (TS), Kripke structures, Petri Nets, ...
- Composition: parallel composition $(P_0 \| P_1)$

Design specifications are often operational.

(1) Design specifications

... express:

- a (set of) potential solution(s) (use a shared variable "turn" ....)
- i.e. specific algorithms/ components/ ...
- can (in principle) be "implemented"

Typical formalisms:

- programs, abstract programs
- (extended) automata, transition systems (TS), Kripke structures, Petri Nets, ...
- Composition: parallel composition ($P_0 \| P_1$)

Design specifications are often operational.

(1) Design specifications

… express:

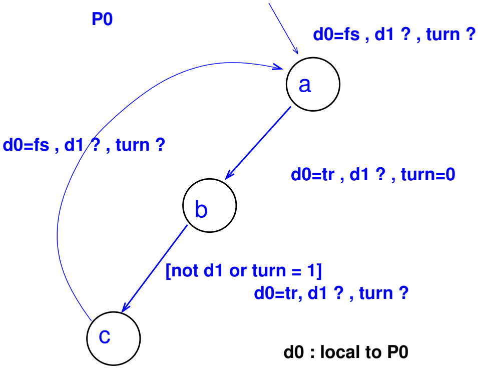
- a (set of) potential solution(s) (use a shared variable “turn” ….)
- i.e. specific algorithms/ components/ …
- can (in principle) be “implemented”

Typical formalisms:

- programs, abstract programs
- (extended) automata, transition systems (TS), Kripke structures, Petri Nets, …
- Composition: parallel composition $(P_0 \| P_1)$

Design specifications are often operational.

Example: Peterson as a composition of symb. TS

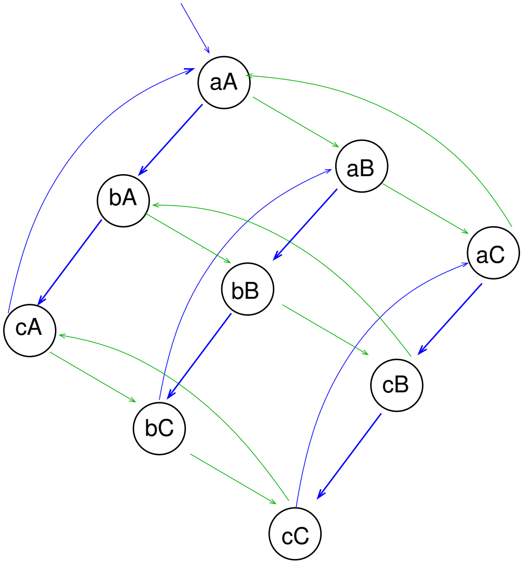


d0 : local to P0

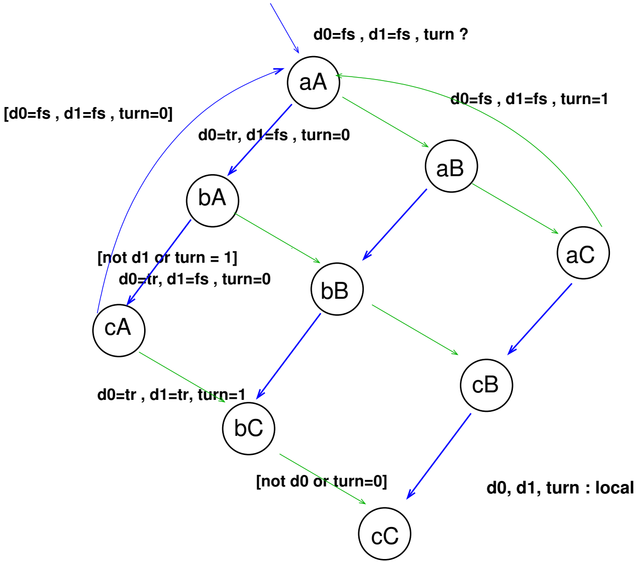
d1 : local to P1

turn : shared

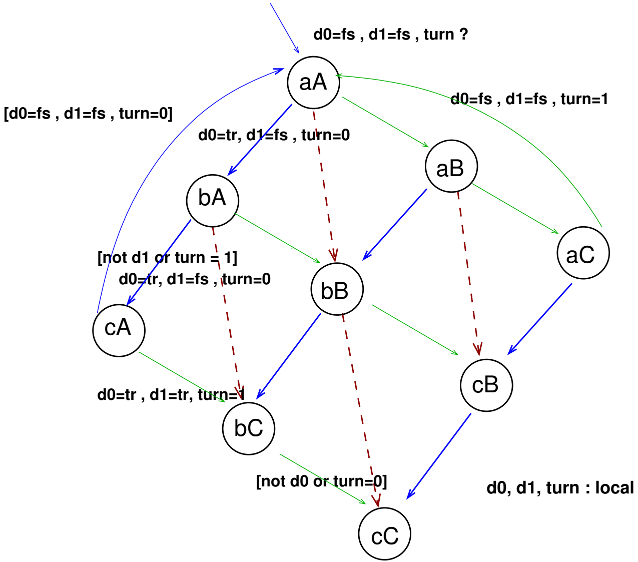
Example: Peterson as a global symbolic TS



Example: Peterson as a global symbolic TS



Example: Peterson as a global symbolic TS



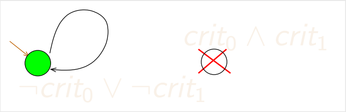
(2) Requirements

… express

- generic or specific properties that a component, system, algorithm, … should have (deadlock freedom, mutual exclusion, …)
- specifies "what" it does, its qualities, not how
- cannot generally not be meaningfully "implemented" (by a compiler)

Typical formalisms:

- (temporal) logic $\forall\Box(\neg crit_0 \vee \neg crit_1)$
- (extended) TS , …



- composition = conjunction *All* requirements must be satisfied

may be "executable" or not.

The difference is more in the intent than the form.

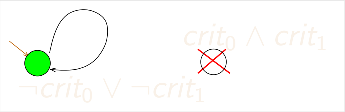
(2) Requirements

… express

- generic or specific properties that a component, system, algorithm, ... should have (deadlock freedom, mutual exclusion, ...)
- specifies "what" it does, its qualities, not how
- cannot generally not be meaningfully "implemented" (by a compiler)

Typical formalisms:

- (temporal) logic $\forall\Box(\neg crit_0 \vee \neg crit_1)$
- (extended) TS , ...



- composition = conjunction *All* requirements must be satisfied

may be "executable" or not.

The difference is more in the intent than the form.

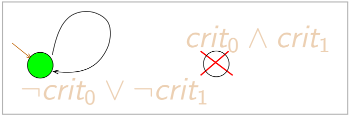
(2) Requirements

… express

- generic or specific properties that a component, system, algorithm, … should have (deadlock freedom, mutual exclusion, …)
- specifies "what" it does, its qualities, not how
- cannot generally not be meaningfully "implemented" (by a compiler)

Typical formalisms:

- (temporal) logic $\forall\Box(\neg crit_0 \vee \neg crit_1)$
- (extended) TS , …



- composition = conjunction *All* requirements must be satisfied

may be "executable" or not.
The difference is more in the intent than the form.

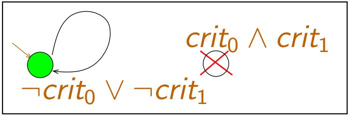
(2) Requirements

... express

- generic or specific properties that a component, system, algorithm, ... should have (deadlock freedom, mutual exclusion, ...)
- specifies “what” it does, its qualities, not how
- cannot generally not be meaningfully “implemented” (by a compiler)

Typical formalisms:

- (temporal) logic $\forall\Box(\neg crit_0 \vee \neg crit_1)$
- (extended) TS , ...



- composition = conjunction *All* requirements must be satisfied

may be “executable” or not.
The difference is more in the intent than the form.

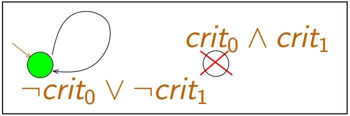
(2) Requirements

... express

- generic or specific properties that a component, system, algorithm, ... should have (deadlock freedom, mutual exclusion, ...)
- specifies "what" it does, its qualities, not how
- cannot generally not be meaningfully "implemented" (by a compiler)

Typical formalisms:

- (temporal) logic $\forall\Box(\neg crit_0 \vee \neg crit_1)$
- (extended) TS , ...



- composition = conjunction *All* requirements must be satisfied

may be "executable" or not.

The difference is more in the intent than the form.

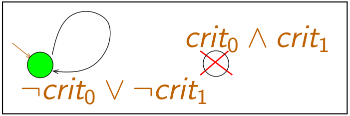
(2) Requirements

… express

- generic or specific properties that a component, system, algorithm, ... should have (deadlock freedom, mutual exclusion, ...)
- specifies “what” it does, its qualities, not how
- cannot generally not be meaningfully “implemented” (by a compiler)

Typical formalisms:

- (temporal) logic $\forall\Box(\neg crit_0 \vee \neg crit_1)$
- (extended) TS , ...



- composition = conjunction *All* requirements must be satisfied

may be “executable” or not.

The difference is more in the intent than the form.

(3) Satisfaction Relations

... defines what is means that M has property φ.

Typically defined on some *semantic domain*:

Semantic Property Domain	Relationship $\models$	Property class
Function relating input/output	equality	(correctness)
Reachable states	inclusion	(invariance)
Sets of executions/prefixes/streams	inclusion	(linear, LTL)
Refusal sets	inclusion	(reactivity)
TS	simulation	(structural)

Verification = checking these relationships

(3) Satisfaction Relations

... defines what is means that M has property φ.

Typically defined on some *semantic domain*:

$$\begin{tikzcd} M,\ \varphi \arrow[d, "\textit{mapping}"] \\ |M|,\ |\varphi| \end{tikzcd}$$

Semantic Property Domain	Relationship $\models$	Property class
Function relating input/output	equality	(correctness)
Reachable states	inclusion	(invariance)

(3) Satisfaction Relations

... defines what is means that M has property φ.

Typically defined on some *semantic domain*:

$$\begin{tikzcd}
M, \varphi \arrow[d, "mapping"] & \\
{|M|, |\varphi|} \arrow[r] & {|M| \models |\varphi|}
\end{tikzcd}$$

Semantic Property Domain	Relationship $\models$	Property class
Function relating input/output	equality	(correctness)
Reachable states	inclusion	(invariance)

(3) Satisfaction Relations

... defines what is means that M has property φ.

Typically defined on some *semantic domain*:

$$\begin{tikzcd}
M,\ \varphi \arrow[d, "mapping"] & & M \models \varphi \\
{|M|,\ |\varphi|} \arrow[rr] & & {|M| \models |\varphi|} \arrow[u, "conclude"']
\end{tikzcd}$$

Semantic Property Domain	Relationship $\models$	Property class
Function relating input/output	equality	(correctness)
Reachable states	inclusion	(invariance)

(3) Satisfaction Relations

... defines what is means that M has property φ.

Typically defined on some *semantic domain*:

$$\begin{tikzcd}
M, \varphi \arrow[d, "mapping"] & & M \models \varphi \\
|M|, |\varphi| \arrow[r] & & |M| \models |\varphi| \arrow[u, "conclude"']
\end{tikzcd}$$

Semantic Property Domain	Relationship $\models$	Property class
Function relating input/output	equality	(correctness)
Reachable states	inclusion	(invariance)
Sets of executions/prefixes/streams	inclusion	(linear, LTL)
Refusal sets	inclusion	(reactivity)
TS	simulation	(structural)

Verification = checking these relationships

(3) Satisfaction Relations

... defines what is means that M has property φ.

Typically defined on some *semantic domain*:

$$\begin{tikzcd} M, \varphi \arrow[d, "mapping"] & M \models \varphi \\ |M|, |\varphi| \arrow[r] & |M| \models |\varphi| \arrow[u, "conclude"'] \end{tikzcd}$$

Semantic Property Domain	Relationship $\models$	Property class
Function relating input/output	equality	(correctness)
Reachable states	inclusion	(invariance)
Sets of executions/prefixes/streams	inclusion	(linear, LTL)
Refusal sets	inclusion	(reactivity)
TS	simulation	(structural)

Verification = checking these relationships

(4) Model-checking

... an algorithm for checking the relation $\models$

$$\models \; : \; \mathcal{M} \times \mathcal{L} \mapsto \{tr, fs, fail\}$$

fail may be due to

- theoretical *undecidability* of $\models$
- excessive complexity (*state explosion*) of the algorithm used
- *incompleteness* of the algorithm

... based on some more or less low-level semantic representation of M, φ

(4) Model-checking

... an algorithm for checking the relation $\models$

$$\models \; : \; \mathcal{M} \times \mathcal{L} \mapsto \{tr, fs, fail\}$$

fail may be due to

- theoretical *undecidability* of $\models$
- excessive complexity (*state explosion*) of the algorithm used
- *incompleteness* of the algorithm

... based on some more or less low-level semantic representation of M, φ

(4) Model-checking

… an algorithm for checking the relation $\models$

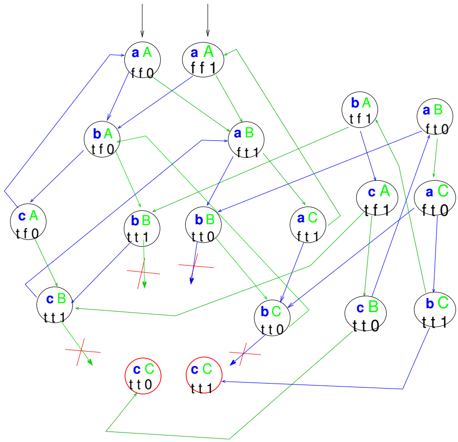
$$\models \;:\; \mathcal{M} \;\times\; \mathcal{L} \;\mapsto\; \{tr, fs, fail\}$$

fail may be due to

- theoretical *undecidability* of $\models$
- excessive complexity (*state explosion*) of the algorithm used
- *incompleteness* of the algorithm

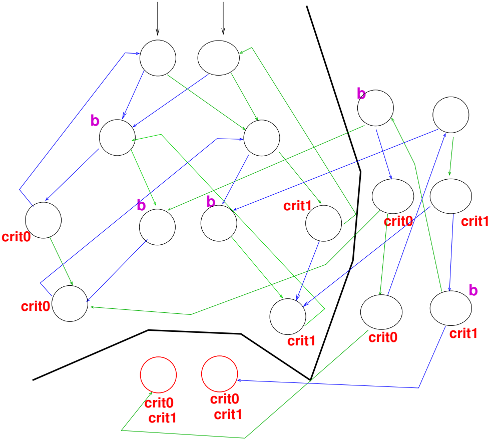
… based on some more or less low-level semantic representation of M, φ

Example: Semantic model of the Peterson algorithm



$|M| \models \varphi$: can be checked easily by calculating fix-points on this graph:

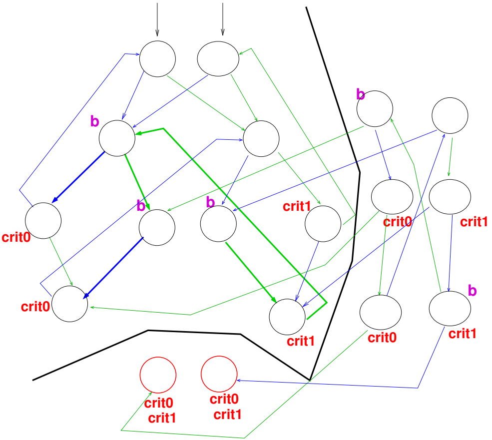
Example: Semantic model of the Peterson algorithm



$|M| \models \varphi$: can be checked easily by calculating fix-points on this graph:

$$init \subseteq |\Box(\neg crit_0 \vee \neg crit_1)|$$

Example: Semantic model of the Peterson algorithm



$|M| \models \varphi$: can be checked easily by calculating fix-points on this graph:

$$b \subseteq |\Diamond crit_0|$$

The main difficulty of verification: complexity

$$\begin{tikzcd}
M, \varphi \arrow[d, "mapping"] & M \models \varphi \quad \text{(yes/no/?)} \\
|M|, |\varphi| \arrow[r] & |M| \models |\varphi| \quad \text{(yes/no/fail)} \arrow[u, "conclude"]
\end{tikzcd}$$

$\mathcal{M}$: compact syntax, program, timed automata, transition system (TS) with variables, composition, ...
$|\mathcal{M}|$: less structure, e.g. labeled TS $\{+$ constraints$\}$

$\Longrightarrow$ Model-checking algorithms $\models$ typically work in 2 steps:

- $\models^{\textit{step } 1}$: transformation into semantic models $|M|$, $|\varphi|$
- $\models^{\textit{step } 2}$: evaluate satisfaction based on $|M|$, $|\varphi|$

Main complexity: *step 1* (for M) $\longrightarrow$ **state explosion**

Note: performant procedures $\models$ mix steps 1 and 2: avoid computing $|M|$ exhaustively. But the problem of complexity explosion remains.

The main difficulty of verification: complexity

$$\begin{tikzcd}
M, \varphi \arrow[d, "mapping"] & & M \models \varphi \quad \text{(yes/no/?)} \\
|M|, |\varphi| \arrow[rr] & & |M| \models |\varphi| \quad \text{(yes/no/fail)} \arrow[u, "conclude"]
\end{tikzcd}$$

$\mathcal{M}$: compact syntax, program, timed automata, transition system (TS) with variables, composition, ...

$|\mathcal{M}|$: less structure, e.g. labeled TS $\{+$ constraints$\}$

$\Longrightarrow$ Model-checking algorithms $\models$ typically work in 2 steps:

- $\models^{step\ 1}$: transformation into semantic models $|M|$, $|\varphi|$
- $\models^{step\ 2}$: evaluate satisfaction based on $|M|$, $|\varphi|$

Main complexity: *step 1* (for M) $\longrightarrow$ **state explosion**

Note: performant procedures $\models$ mix steps 1 and 2: avoid computing $|M|$ exhaustively. But the problem of complexity explosion remains.

The main difficulty of verification: complexity

$$\begin{tikzcd}
M, \varphi \arrow[d, "mapping"] & M \models \varphi & \text{(yes/no/?)} \\
{|M|, |\varphi|} \arrow[r] & {|M| \models |\varphi|} & \text{(yes/no/fail)} \arrow[u, "conclude"']
\end{tikzcd}$$

$\mathcal{M}$: compact syntax, program, timed automata, transition system (TS) with variables, composition, ...
$|\mathcal{M}|$: less structure, e.g. labeled TS $\{+$ constraints$\}$

$\Longrightarrow$ Model-checking algorithms $\models$ typically work in 2 steps:

- $\models^{step\ 1}$: transformation into semantic models $|M|$, $|\varphi|$
- $\models^{step\ 2}$: evaluate satisfaction based on $|M|$, $|\varphi|$

Main complexity: *step 1* (for M) $\longrightarrow$ **state explosion**

Note: performant procedures $\models$ mix steps 1 and 2: avoid computing $|M|$ exhaustively. But the problem of complexity explosion remains.

The main difficulty of verification: complexity

$$\begin{tikzcd}
M, \varphi \arrow[d, "mapping"] & & M \models \varphi & \text{(yes/no/?)} \\
{|M|, |\varphi|} \arrow[rr] & & {|M| \models |\varphi|} \arrow[u, "conclude"'] & \text{(yes/no/fail)}
\end{tikzcd}$$

$\mathcal{M}$: compact syntax, program, timed automata, transition system (TS) with variables, composition, ...
$|\mathcal{M}|$: less structure, e.g. labeled TS $\{+ \text{ constraints}\}$

$\Longrightarrow$ Model-checking algorithms $\models$ typically work in 2 steps:

- $\models^{\textit{step } 1}$: transformation into semantic models $|M|$, $|\varphi|$
- $\models^{\textit{step } 2}$: evaluate satisfaction based on $|M|$, $|\varphi|$

Main complexity: *step 1* (for M) $\longrightarrow$ **state explosion**

Note: performant procedures $\models$ mix steps 1 and 2: avoid computing $|M|$ exhaustively. But the problem of complexity explosion remains.

The main difficulty of verification: complexity

$$\begin{tikzcd}
M, \varphi \arrow[d, "mapping"] & & M \models \varphi & \text{(yes/no/?)} \\
{|M|, |\varphi|} \arrow[rr] & & {|M| \models |\varphi|} \arrow[u, "conclude"'] & \text{(yes/no/fail)}
\end{tikzcd}$$

$\mathcal{M}$: compact syntax, program, timed automata, transition system (TS) with variables, composition, ...
$|\mathcal{M}|$: less structure, e.g. labeled TS $\{+$ constraints$\}$

$\Longrightarrow$ Model-checking algorithms $\models$ typically work in 2 steps:

- $\models^{step\ 1}$: transformation into semantic models $|M|$, $|\varphi|$
- $\models^{step\ 2}$: evaluate satisfaction based on $|M|$, $|\varphi|$

Main complexity: *step 1* (for M) $\longrightarrow$ **state explosion**

Note: performant procedures $\models$ mix steps 1 and 2: avoid computing $|M|$ exhaustively. But the problem of complexity explosion remains.

The main difficulty of verification: complexity

$$\begin{tikzcd}
M, \varphi \arrow[d, "mapping"] & & M \models \varphi \quad \text{(yes/no/?)} \\
|M|, |\varphi| \arrow[rr] & & |M| \models |\varphi| \quad \text{(yes/no/fail)} \arrow[u, "conclude"]
\end{tikzcd}$$

$\mathcal{M}$: compact syntax, program, timed automata, transition system (TS) with variables, composition, ...
$|\mathcal{M}|$: less structure, e.g. labeled TS $\{+$ constraints$\}$

$\Longrightarrow$ Model-checking algorithms $\models$ typically work in 2 steps:

- $\models^{\mathit{step}\ 1}$: transformation into semantic models $|M|$, $|\varphi|$
- $\models^{\mathit{step}\ 2}$: evaluate satisfaction based on $|M|$, $|\varphi|$

Main complexity: *step 1* (for M) $\longrightarrow$ **state explosion**

Note: performant procedures $\models$ mix steps 1 and 2: avoid computing $|M|$ exhaustively. But the problem of complexity explosion remains.

The main difficulty of verification: complexity

$$\begin{tikzcd}
M, \varphi \arrow[d, "mapping"] & M \models \varphi & \text{(yes/no/?)} \\
|M|, |\varphi| \arrow[r] & |M| \models |\varphi| \arrow[u, "conclude"'] & \text{(yes/no/fail)}
\end{tikzcd}$$

$\mathcal{M}$: compact syntax, program, timed automata, transition system (TS) with variables, composition, ...
$|\mathcal{M}|$: less structure, e.g. labeled TS $\{+$ constraints$\}$

$\Longrightarrow$ Model-checking algorithms $\models$ typically work in 2 steps:

- $\models^{\textit{step } 1}$: transformation into semantic models $|M|$, $|\varphi|$
- $\models^{\textit{step } 2}$: evaluate satisfaction based on $|M|$, $|\varphi|$

Main complexity: *step 1* (for M) $\longrightarrow$ **state explosion**

Note: performant procedures $\models$ mix steps 1 and 2: avoid computing $|M|$ exhaustively. But the problem of complexity explosion remains.

What is abstraction?

An **abstraction** is a *property preserving transformation*

Given

- a verification setting: $(\mathcal{M}, \mathcal{L}, \models)$
- a transformation α : $\mathcal{M} \mapsto \mathcal{M}^A$ with $\mathcal{M}^A \subseteq \mathcal{M}$

Then

- α is an *abstraction* for $(\mathcal{M}, \mathcal{L}, \models)$ if

$$\forall M \in \mathcal{M}\ \forall \varphi \in \mathcal{L} \,.\, \alpha(M) \models \varphi \text{ implies } M \models \varphi$$

- α is called *strongly property preserving* if in addition

$$\forall M \in \mathcal{M}\ \forall \varphi \in \mathcal{L} \,.\, \alpha(M) \not\models \varphi \text{ implies } M \not\models \varphi$$

What is abstraction?

An **abstraction** is a *property preserving transformation*

Given

- a verification setting: $(\mathcal{M}, \mathcal{L}, \models)$
- a transformation $\alpha : \mathcal{M} \mapsto \mathcal{M}^A$ with $\mathcal{M}^A \subseteq \mathcal{M}$

Then

- α is an *abstraction* for $(\mathcal{M}, \mathcal{L}, \models)$ if

$$\forall M \in \mathcal{M}\ \forall \varphi \in \mathcal{L}\ .\ \alpha(M) \models \varphi \text{ implies } M \models \varphi$$

- α is called *strongly property preserving* if in addition

$$\forall M \in \mathcal{M}\ \forall \varphi \in \mathcal{L}\ .\ \alpha(M) \not\models \varphi \text{ implies } M \not\models \varphi$$

What is abstraction?

An **abstraction** is a *property preserving transformation*

Given

- a verification setting: $(\mathcal{M}, \mathcal{L}, \models)$
- a transformation α : $\mathcal{M} \mapsto \mathcal{M}^A$ with $\mathcal{M}^A \subseteq \mathcal{M}$

Then

- α is an *abstraction* for $(\mathcal{M}, \mathcal{L}, \models)$ if

$$\forall M \in \mathcal{M}\ \forall \varphi \in \mathcal{L} .\ \alpha(M) \models \varphi \text{ implies } M \models \varphi$$

- α is called *strongly property preserving* if in addition

$$\forall M \in \mathcal{M}\ \forall \varphi \in \mathcal{L} .\ \alpha(M) \not\models \varphi \text{ implies } M \not\models \varphi$$

What is abstraction?

An **abstraction** is a *property preserving transformation*

Given

- a verification setting: $(\mathcal{M}, \mathcal{L}, \models)$
- a transformation $\alpha : \mathcal{M} \mapsto \mathcal{M}^A$ with $\mathcal{M}^A \subseteq \mathcal{M}$

Then

- α is an *abstraction* for $(\mathcal{M}, \mathcal{L}, \models)$ if

$$\forall M \in \mathcal{M}\ \forall \varphi \in \mathcal{L}\ .\ \alpha(M) \models \varphi \text{ implies } M \models \varphi$$

- α is called *strongly property preserving* if in addition

$$\forall M \in \mathcal{M}\ \forall \varphi \in \mathcal{L}\ .\ \alpha(M) \not\models \varphi \text{ implies } M \not\models \varphi$$

What is abstraction?

An **abstraction** is a *property preserving transformation*

Given

- a verification setting: $(\ \mathcal{M}, \mathcal{L}, \models)$
- a transformation $\alpha\ :\ \mathcal{M} \mapsto \mathcal{M}^A$ with $\mathcal{M}^A \subseteq \mathcal{M}$

Then

- α is an *abstraction* for $(\mathcal{M}, \mathcal{L}, \models)$ if

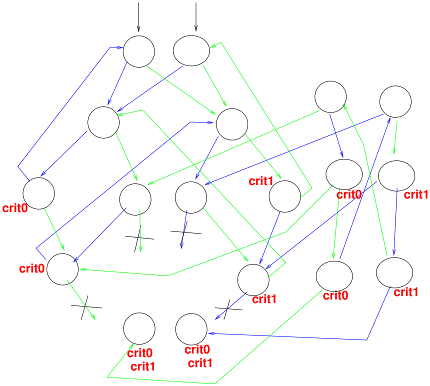
$$\forall M \in \mathcal{M}\ \forall \varphi \in \mathcal{L}\ .\ \alpha(M) \models \varphi \text{ implies } M \models \varphi$$

- α is called *strongly property preserving* if in addition

$$\forall M \in \mathcal{M}\ \forall \varphi \in \mathcal{L}\ .\ \alpha(M) \not\models \varphi \text{ implies } M \not\models \varphi$$

Example: Abstracted semantic model of Peterson

α: (1) group *states* to 5 abstract ones (black, green, blue, red, yellow), (2) draw a (green / blue) *transition* between abstract states if there is one between a corresponding pair of concrete ones



$\alpha(|M|)$ satisfies properties (1) *mutual exclusion* and (2) *non-blocking*. We cannot evaluate (3) *fairness*.

What can we conclude?

Answer: only (1) allows a conclusion for the original concrete model. α does not preserve the other results

Example: Abstracted semantic model of Peterson

α: (1) group *states* to 5 abstract ones (black, green, blue, red, yellow),
(2) draw a (green / blue) *transition* between abstract states if there is one between a corresponding pair of concrete ones

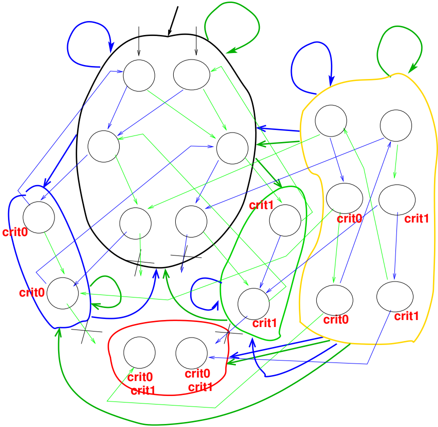
$\alpha(|M|)$ satisfies properties (1) *mutual exclusion* and (2) *non-blocking*. We cannot evaluate (3) *fairness*.

What can we conclude?

Answer: only (1) allows a conclusion for the original concrete model. α does not preserve the other results

Example: Abstracted semantic model of Peterson

α: (1) group *states* to 5 abstract ones (black, green, blue, red, yellow), (2) draw a (green / blue) *transition* between abstract states if there is one between a corresponding pair of concrete ones



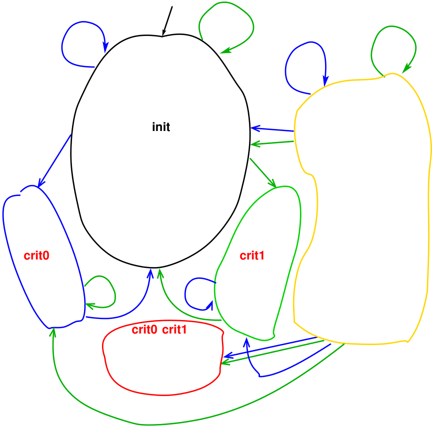
$\alpha(|M|)$ satisfies properties (1) *mutual exclusion* and (2) *non-blocking*. We cannot evaluate (3) *fairness*.

What can we conclude?

Answer: only (1) allows a conclusion for the original concrete model. α does not preserve the other results

Example: Abstracted semantic model of Peterson

α: (1) group *states* to 5 abstract ones (black, green, blue, red, yellow),
(2) draw a (green / blue) *transition* between abstract states if there is one between a corresponding pair of concrete ones



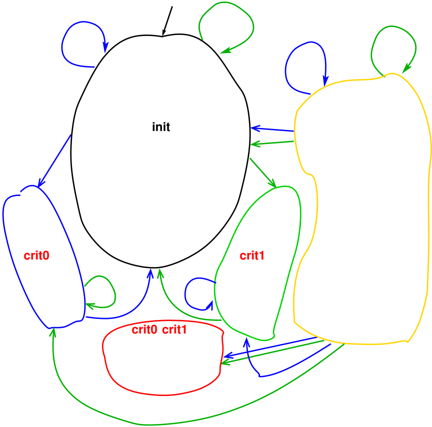
$\alpha(|M|)$ satisfies properties (1) *mutual exclusion* and (2) *non-blocking*. We cannot evaluate (3) *fairness*.

What can we conclude?

Answer: only (1) allows a conclusion for the original concrete model. α does not preserve the other results

Example: Abstracted semantic model of Peterson

α: (1) group *states* to 5 abstract ones (black, green, blue, red, yellow), (2) draw a (green / blue) *transition* between abstract states if there is one between a corresponding pair of concrete ones



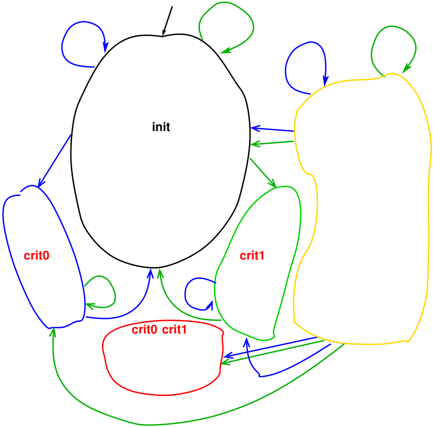
$\alpha(|M|)$ satisfies properties (1) *mutual exclusion* and (2) *non-blocking*. We cannot evaluate (3) *fairness*.

What can we conclude?

Answer: only (1) allows a conclusion for the original concrete model. α does not preserve the other results

Example: Abstracted semantic model of Peterson

α: (1) group *states* to 5 abstract ones (black, green, blue, red, yellow), (2) draw a (green / blue) *transition* between abstract states if there is one between a corresponding pair of concrete ones



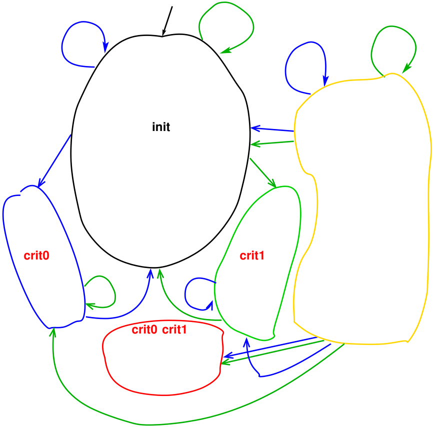
$\alpha(|M|)$ satisfies properties (1) *mutual exclusion* and (2) *non-blocking*. We cannot evaluate (3) *fairness*.

What can we conclude?

Answer: only (1) allows a conclusion for the original concrete model. α does not preserve the other results

Example: Abstracted semantic model of Peterson

α: (1) group *states* to 5 abstract ones (black, green, blue, red, yellow),
(2) draw a (green / blue) *transition* between abstract states if there is one between a corresponding pair of concrete ones



$\alpha(|M|)$ satisfies properties (1) *mutual exclusion* and (2) *non-blocking*. We cannot evaluate (3) *fairness*.

What can we conclude?

Answer: only (1) allows a conclusion for the original concrete model. α does not preserve the other results

Why abstraction? what have we gained?

The main motivation: avoid *state explosion*

Is our abstraction for Peterson useful?: not really,

Note that our abstraction of Peterson cannot be obtained that way.

Why abstraction? what have we gained?

The main motivation: avoid *state explosion*

Is our abstraction for Peterson useful?: not really,

Note that our abstraction of Peterson cannot be obtained that way.

Why abstraction? what have we gained?

The main motivation: avoid *state explosion*

Is our abstraction for Peterson useful?: not really,

Note that our abstraction of Peterson cannot be obtained that way.

Why abstraction? what have we gained?

The main motivation: avoid *state explosion*

Is our abstraction for Peterson useful?: not really,

$$\begin{tikzcd}
M,\ \varphi \arrow[d, "mapping"] & \\
\lvert M \rvert \arrow[r, "\alpha"] & \lvert M^A \rvert
\end{tikzcd}$$

Note that our abstraction of Peterson cannot be obtained that way.

Why abstraction? what have we gained?

The main motivation: avoid *state explosion*

Is our abstraction for Peterson useful?: not really,

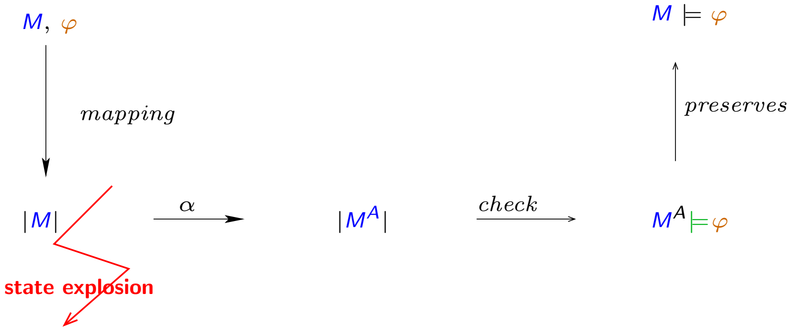
$$\begin{tikzcd}
M, \varphi \arrow[d, "mapping"] & & M \models \varphi \\
|M| \arrow[r, "\alpha"] & |M^A| \arrow[r, "check"] & M^A \models \varphi \arrow[u, "preserves"']
\end{tikzcd}$$

Note that our abstraction of Peterson cannot be obtained that way.

Why abstraction? what have we gained?

The main motivation: avoid *state explosion*

Is our abstraction for Peterson useful?: not really,



$$\begin{tikzcd}
M,\ \varphi \arrow[d, "mapping"] & & M \models \varphi \\
|M| \arrow[r, "\alpha"] & |M^A| \arrow[r, "check"] & M^A \models \varphi \arrow[u, "preserves"']
\end{tikzcd}$$

state explosion

Note that our abstraction of Peterson cannot be obtained that way.

Why abstraction? what have we gained?

The main motivation: avoid *state explosion*

Is our abstraction for Peterson useful?: not really,

$$\begin{tikzcd}
M, \varphi \arrow[dr] & & M \models \varphi \\
& |M^A| \arrow[r, "check"] & M^A \models \varphi \arrow[u, "preserves"']
\end{tikzcd}$$

Note that our abstraction of Peterson cannot be obtained that way.

Why abstraction? what have we gained?

The main motivation: avoid *state explosion*

Is our abstraction for Peterson useful?: not really,

$$\begin{tikzcd}
M, \varphi \arrow[r, dashed] \arrow[dr] & M^A \arrow[d, dashed, "mapping"] & M \models \varphi \\
 & \lvert M^A \rvert \arrow[r, "check"] & M^A \models \varphi \arrow[u, "preserves"']
\end{tikzcd}$$

Note that our abstraction of Peterson cannot be obtained that way.

Why abstraction? what have we gained?

The main motivation: avoid *state explosion*

Is our abstraction for Peterson useful?: not really,

$$\begin{tikzcd}
M, \varphi \arrow[r, dashed] \arrow[dr] & M^A \arrow[d, dashed, "mapping"] & M \models \varphi \\
& {|M^A|} \arrow[r, "check"] & M^A \models \varphi \arrow[u, "preserves"']
\end{tikzcd}$$

Note that our abstraction of Peterson cannot be obtained that way.

What is particular to system verification?

We are interested in closed systems (the system + some more or less specific environment).

A system is composed of a (large) number of components (in parallel, $\|$) and there are requirements related to desired and undesired global (*emergent*) properties.

Usual verification settings guarantee:

What is particular to system verification?

We are interested in closed systems (the system + some more or less specific environment).

A system is composed of a (large) number of components (in parallel, $\|$) .

Usual verification settings guarantee:

$$M_1 \quad \| \quad M_2 \quad \| \qquad\qquad \| \quad M_n \quad \models \quad \varphi^{glob}$$

What is particular to system verification?

We are interested in closed systems (the system + some more or less specific environment).

A system is composed of a (large) number of components (in parallel, $\|$) .

Usual verification settings guarantee:

$$\begin{tikzcd}
M_1 \arrow[dd] & \| & M_2 \arrow[dd] & \| & & \| & M_n \arrow[dd] \\
 & & & \alpha & & & \\
M_1^A & \| & M_2^A & \| & & \| & M_n^A
\end{tikzcd}$$

What is particular to system verification?

We are interested in closed systems (the system + some more or less specific environment).

A system is composed of a (large) number of components (in parallel, $\|$) .

Usual verification settings guarantee:

$$\begin{tikzcd}
M_1 \arrow[d] & \| & M_2 \arrow[d, "\alpha"] & \| & & \| & M_n \arrow[d] & & \\
\xcancel{M_1^A} & \| & M_2^A & \| & & \| & \xcancel{M_n^A} & \models & \varphi^{glob}
\end{tikzcd}$$

What is particular to system verification?

We are interested in closed systems (the system + some more or less specific environment).

A system is composed of a (large) number of components (in parallel, $\|$) .

Usual verification settings guarantee:

$$\begin{tikzcd}
M_1 & \| & M_2 \arrow[d, magenta, "\alpha"] & \| & & \| & M_n \arrow[d] & \models & \varphi^{glob} \\
M_1^A \arrow[from=u] & \| & M_2^A & \| & & \| & M_n^A & \models \arrow[u, "conclude"'] & \varphi^{glob}
\end{tikzcd}$$

What is particular to system verification?

We are interested in closed systems (the system + some more or less specific environment).

A system is composed of a (large) number of components (in parallel, $\|$) .

Usual verification settings guarantee:

$$M_1 \quad \| \quad \boxed{M_2} \quad \| \qquad\qquad \| \quad M_n \quad \models \quad \varphi^2$$

What is particular to system verification?

We are interested in closed systems (the system + some more or less specific environment).

A system is composed of a (large) number of components (in parallel, $\|$) .

Usual verification settings guarantee:

$$\begin{tikzcd}
M_1 & \| & \boxed{M_2} \arrow[d, "\alpha"] & \| & & \| & M_n \\
& & M_2^A & & & &
\end{tikzcd}$$

What is particular to system verification?

We are interested in closed systems (the system + some more or less specific environment).

A system is composed of a (large) number of components (in parallel, $\|$) .

Usual verification settings guarantee:

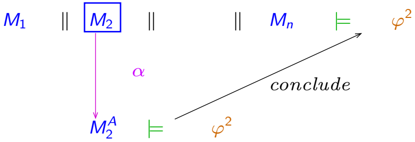
$$\begin{tikzcd}
M_1 & \| & \boxed{M_2} \arrow[d, "\alpha"] & \| & & \| & M_n \\
& & M_2^A & \models & \varphi^2 & &
\end{tikzcd}$$

What is particular to system verification?

We are interested in closed systems (the system + some more or less specific environment).

A system is composed of a (large) number of components (in parallel, $\|$) .

Usual verification settings guarantee:



$M_1 \quad \| \quad \boxed{M_2} \quad \| \qquad\qquad \| \quad M_n \quad \models \quad \varphi^2$

α

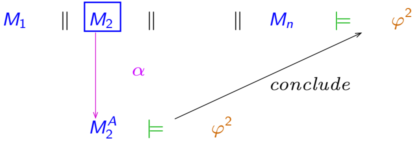
$conclude$

$M_2^A \quad \models \quad \varphi^2$

What is particular to system verification?

A system is composed of a (large) number of components (in parallel, $\|$) and there are requirements related to desired and undesired global (*emergent*) properties.

Usual settings guarantee



But — it is hard to find a useful α for which the premises $\alpha(M_1)\|\ldots\|\alpha(M_n)\models\varphi$ or $\alpha(M_i)\models\varphi^i$ hold and can be checked.

We want appropriate reasoning rules for

- composing verification results
- combining them with abstraction

What is particular to system verification?

A system is composed of a (large) number of components (in parallel, $\|$) and there are requirements related to desired and undesired global (*emergent*) properties.

Usual settings guarantee

$$\begin{tikzcd}
M_1 & \| & \boxed{M_2} \arrow[d, "\alpha"] & \| & & \| & M_n & \models & \varphi^2 \\
 & & M_2^A & \models \arrow[urrrrr, "conclude"'] & \varphi^2 & & & &
\end{tikzcd}$$

But — it is hard to find a useful α for which the premises $\alpha(M_1)\|...\|\alpha(M_n)\models\varphi$ or $\alpha(M_i)\models\varphi^i$ hold and can be checked.

We want appropriate reasoning rules for

- composing verification results
- combining them with abstraction

Summary: problems we do address

- *Property preservation* (which abstraction preserves which properties)
- How to *effectively calculate* abstractions
- How to achieve *verification of global properties* by combining abstraction and *rules for composing results*

Summary: problems we do address

- *Property preservation* (which abstraction preserves which properties)
- How to *effectively calculate* abstractions
- How to achieve *verification of global properties* by combining abstraction and *rules for composing results*

Summary: problems we do address

- *Property preservation* (which abstraction preserves which properties)
- How to *effectively calculate* abstractions
- How to achieve *verification of global properties* by combining abstraction and *rules for composing results*

Summary: problems we do address

- *Property preservation* (which abstraction preserves which properties)
- How to *effectively calculate* abstractions
- How to achieve *verification of global properties* by combining abstraction and *rules for composing results*

Summary: problems we do not address

- Adequate *languages* for expressing models and requirements.
- Appropriate *composition* frameworks
- Appropriate satisfaction relations $\models$
- Algorithms $\models$ for solving verification problems $M \models \varphi$ for a given framework.
- Abstraction refinement: when both $M^A \models \varphi$ and $M^A \not\models \varphi$ fail (CEGAR approaches).