An *Informal* Introduction to

Formal Methods for Software Engineering

May 3, 2019

Luigi Libero Lucio Starace

Università degli Studi di Napoli Federico II

Agenda

Software Verification

Agenda

Software Verification

Formal Methods
- Formal Specification
- Formal Verification

Agenda

Agenda

- Software Verification
- Formal Methods
  - Formal Specification
  - Formal Verification
- Formal Methods in Software Engineering
- Practice time!

Agenda

Software Verification

Formal Methods
- Formal Specification
- Formal Verification

Formal Methods in Software Engineering

Practice time!

Take Home Messages

Software Verification

Software Verification
What it's all about

Software Verification

The process of checking that a system meets certain requirements derived from a given *specification*.

Software Verification
What it's all about

Software Verification

The process of checking that a system meets certain requirements derived from a given *specification*.

Why should we care?

Software Verification
What it's all about

Software Verification

The process of checking that a system meets certain requirements derived from a given *specification*.

Why should we care?

- Computer systems are everywhere and we depend more and more on them;

Software Verification
What it's all about

Software Verification

The process of checking that a system meets certain requirements derived from a given *specification*.

Why should we care?

- Computer systems are everywhere and we depend more and more on them;
- Malfunctions may cause financial losses.

Software Verification
What it's all about

Software Verification

The process of checking that a system meets certain requirements derived from a given *specification*.

Why should we care?

- Computer systems are everywhere and we depend more and more on them;
- Malfunctions may cause financial losses **or worse!**

Software Verification
Classic techniques

- Software Testing

Software Verification
Classic techniques

- Software Testing
  - dynamic analysis (software execution involved);

Software Verification
Classic techniques

- Software Testing
  - dynamic analysis (software execution involved);
  - a suite of *test cases*, each specifying inputs and expected system behaviour, is typically produced by software testers.

Software Verification
Classic techniques

- Software Testing
  - dynamic analysis (software execution involved);
  - a suite of *test cases*, each specifying inputs and expected system behaviour, is typically produced by software testers.
- Code inspection

Software Verification
Classic techniques

- Software Testing
  - dynamic analysis (software execution involved);
  - a suite of *test cases*, each specifying inputs and expected system behaviour, is typically produced by software testers.
- Code inspection
  - static analysis (no software execution involved);

Software Verification
Classic techniques

- Software Testing
  - dynamic analysis (software execution involved);
  - a suite of *test cases*, each specifying inputs and expected system behaviour, is typically produced by software testers.
- Code inspection
  - static analysis (no software execution involved);
  - careful scrutiny of the source code carried on by software engineers.

Software Verification

When classic techniques fall short

Testing and code inspection are **very** effective at detecting bugs.

Software Verification
When classic techniques fall short

Testing and code inspection are **very** effective at detecting bugs, but...

- cannot prove their absence;

> [...] *program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence.*
>
> – *The humble programmer*, E. W. Dijkstra [Dij72]

Software Verification
When classic techniques fall short

Testing and code inspection are **very** effective at detecting bugs, but...

- cannot prove their absence;
- ineffective with concurrent systems;

> [...] *a concurrent program can withstand very careful scrutiny without revealing its errors. The only way we can be sure that a concurrent program does what we think it does is to prove rigorously that it does it.*
>
> – *Proving liveness properties of concurrent programs*, L. Lamport [LO82]

Software Verification
When classic techniques fall short

Testing and code inspection are **very** effective at detecting bugs, but…

- cannot prove their absence;
- ineffective with concurrent systems;
- expensive and time-consuming.

Software Verification
When classic techniques fall short

Testing and code inspection are **very** effective at detecting bugs, but...

- cannot prove their absence;
- ineffective with concurrent systems;
- expensive and time-consuming.
- only feasible in later stages of the software lifecycle;

Software Verification
When classic techniques fall short

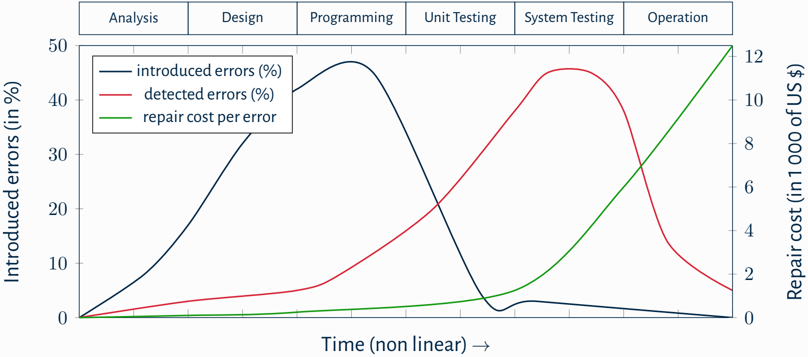


Figure: Error introduction, detection, and repair costs [BK08]

Formal Methods

Formal Methods

Formal Methods [BK08]

Formal methods can be considered as the applied mathematics for modelling and analyzing ICT systems.

Formal Methods

Formal Methods [BK08]

Formal methods can be considered as the applied mathematics for modelling and analyzing ICT systems.

- Formal Specification

Formal Methods

Formal Methods [BK08]

Formal methods can be considered as the applied mathematics for modelling and analyzing ICT systems.

- Formal Specification
  - system modelling languages;

Formal Methods

Formal Methods [BK08]

Formal methods can be considered as the applied mathematics for modelling and analyzing ICT systems.

- Formal Specification
  - system modelling languages;
  - property specification languages.

Formal Methods

Formal Methods [BK08]

Formal methods can be considered as the applied mathematics for modelling and analyzing ICT systems.

- Formal Specification
  - system modelling languages;
  - property specification languages.
- Formal Verification

Formal Methods

Formal Methods [BK08]

Formal methods can be considered as the applied mathematics for modelling and analyzing ICT systems.

- Formal Specification
  - system modelling languages;
  - property specification languages.
- Formal Verification
  - deductive verification (theorem proving);

Formal Methods

Formal Methods [BK08]

Formal methods can be considered as the applied mathematics for modelling and analyzing ICT systems.

- Formal Specification
  - system modelling languages;
  - property specification languages.
- Formal Verification
  - deductive verification (theorem proving);
  - automatic verification (model checking).

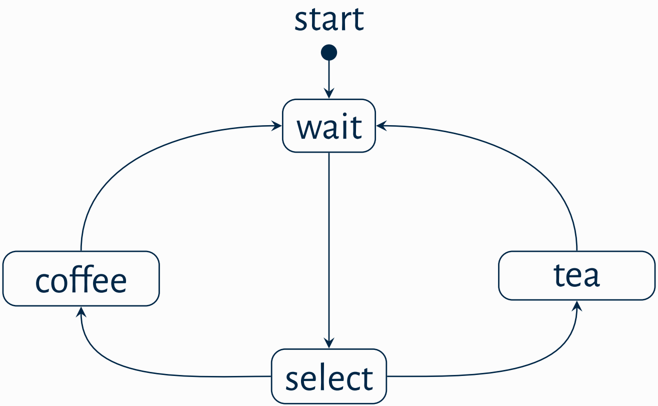
Formal Methods

Formal Methods [BK08]

Formal methods can be considered as the applied mathematics for modelling and analyzing ICT systems.

- Formal Specification
  - system modelling languages;
  - property specification languages.
- Formal Verification
  - deductive verification (theorem proving);
  - automatic verification (model checking).
- Others (formal synthesis)

Formal Specification: Models
Transition Systems (TS)



- the set of states is called *state space*.

Formal Specification: Models
Modelling Languages: features

- Precise and Unambiguous;

Formal Specification: Models
Modelling Languages: features

- Precise and Unambiguous;
    - formally-defined syntax and semantics;

Formal Specification: Models
Modelling Languages: features

- Precise and Unambiguous;
  - formally-defined syntax and semantics;
- "As simple as possible, as rich as needed" [Gli]

Formal Specification: Models
Modelling Languages: features

- Precise and Unambiguous;
  - formally-defined syntax and semantics;
- "As simple as possible, as rich as needed" [Gli]
  - describe relevant aspects in a "natural" way;

Formal Specification: Models
Modelling Languages: features

- Precise and Unambiguous;
  - formally-defined syntax and semantics;
- "As simple as possible, as rich as needed" [Gli]
  - describe relevant aspects in a "natural" way;
  - trade-off between expressivity and analysis complexity;

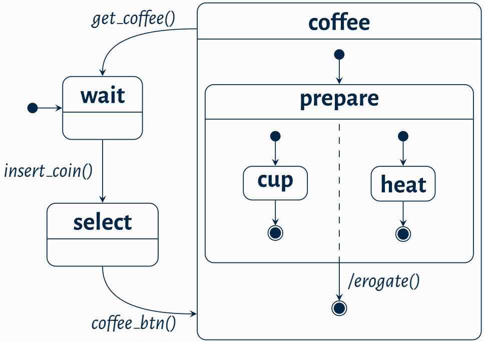
Formal Specification: Models
Modelling Languages: features

- Precise and Unambiguous;
  - formally-defined syntax and semantics;
- “As simple as possible, as rich as needed” [Gli]
  - describe relevant aspects in a “natural” way;
  - trade-off between expressivity and analysis complexity;
  - using TS to model complex systems may be a bad idea: often higher-level languages are used instead.

Formal Specification: Models

Higher-level Modelling Languages: examples

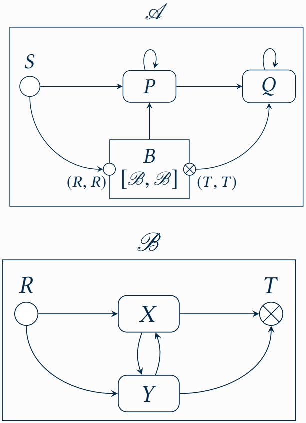
- Statecharts;



Formal Specification: Models

Higher-level Modelling Languages: examples

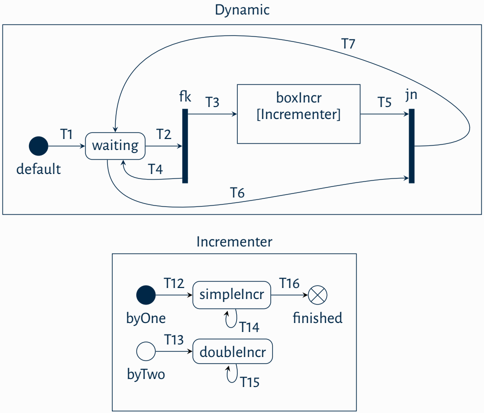
- Statecharts;
- Hierarchical Machines;



Formal Specification: Models

Higher-level Modelling Languages: examples

- Statecharts;
- Hierarchical Machines;
- Dynamic State Machines;



Formal Specification: Models

Higher-level Modelling Languages: examples

- Statecharts;
- Hierarchical Machines;
- Dynamic State Machines;
- PROMELA.

```
active proctype A(){
  do
  :: (1) -> a=0;
  :: (1) -> run B();
  od
}

proctype B() {
  /*...*/
}
```

Formal Specification: Models

Higher-level Modelling Languages: examples

- Statecharts;
- Hierarchical Machines;
- Dynamic State Machines;
- Promela.

Semantics can be defined in terms of transition systems.

```
active proctype A(){
  do
  :: (1) -> a=0;
  :: (1) -> run B();
  od
}

proctype B() {
  /*...*/
}
```

Formal Specification: Models

Higher-level Modelling Languages: examples

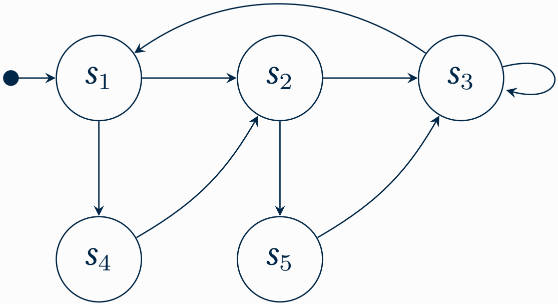
- Statecharts[1];
- Hierarchical Machines[2];
- Dynamic State Machines[3];
- PROMELA[4].

Semantics can be defined in terms of transition systems.

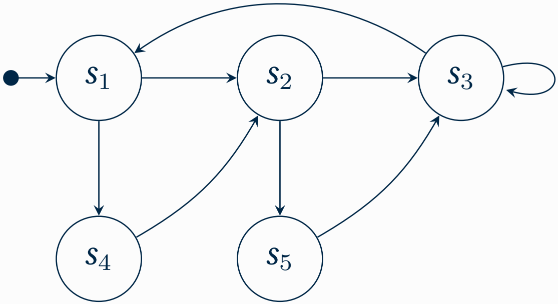
```
active proctype A(){
  do
  :: (1) -> a=0;
  :: (1) -> run B();
  od
}

proctype B() {
  /*...*/
}
```

1 see Harel et al., [Har87; HN96]
2 see Alur et al., [AKY99]
3 see Benerecetti et al., [Ben+17]
4 see [PRO]

Formal Specification: Properties
System Behaviours



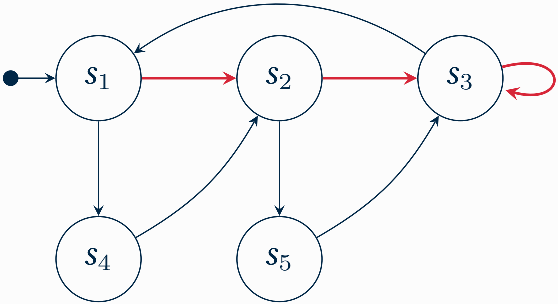
Formal Specification: Properties
System Behaviours



Possible behaviours:

Formal Specification: Properties

System Behaviours

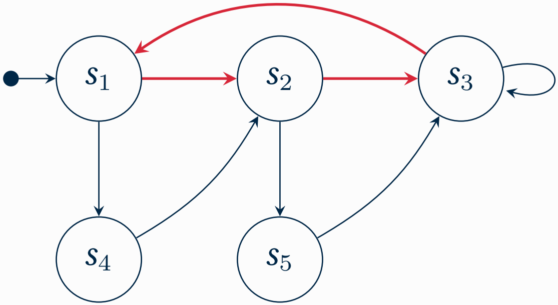


Possible behaviours:

- $\pi_1 = s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_3 \rightarrow s_3 \rightarrow s_3 \rightarrow \dots$ $\qquad s_1\, s_2\, (s_3)^\omega$

Formal Specification: Properties

System Behaviours

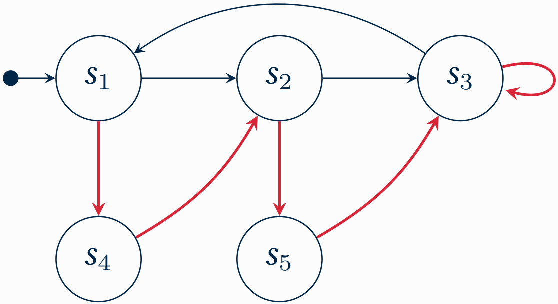


Possible behaviours:

- $\pi_1 = s_1 \to s_2 \to s_3 \to s_3 \to s_3 \to s_3 \to \dots$ $\quad s_1\, s_2\, (s_3)^\omega$
- $\pi_2 = s_1 \to s_2 \to s_3 \to s_1 \to s_2 \to s_3 \to \dots$ $\quad (s_1\, s_2\, s_3)^\omega$

Formal Specification: Properties

System Behaviours



Possible behaviours:

- $\pi_1 = s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_3 \rightarrow s_3 \rightarrow s_3 \rightarrow \dots$ $\qquad s_1\, s_2\, (s_3)^\omega$
- $\pi_2 = s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \dots$ $\qquad (s_1\, s_2\, s_3)^\omega$
- $\pi_3 = s_1 \rightarrow s_4 \rightarrow s_2 \rightarrow s_5 \rightarrow s_3 \rightarrow s_3 \rightarrow \dots$ $\qquad s_1\, s_4\, s_2\, s_5\, (s_3)^\omega$

Formal Specification

Temporal logics: timeline

One way of formally specifying properties of behaviours is using **temporal logics**.

Formal Specification
Temporal logics: timeline

One way of formally specifying properties of behaviours is using **temporal logics**.

A great deal of temporal logics have been proposed in the literature:

Formal Specification

Temporal logics: timeline

One way of formally specifying properties of behaviours is using **temporal logics**.

A great deal of temporal logics have been proposed in the literature:

- **LTL** (Linear-time Temporal Logic) was introduced by Pnueli in 1977 [Pnu77];

Formal Specification
Temporal logics: timeline

One way of formally specifying properties of behaviours is using **temporal logics**.

A great deal of temporal logics have been proposed in the literature:

- **LTL** (Linear-time Temporal Logic) was introduced by Pnueli in 1977 [Pnu77];
- **CTL**, **CTL*** (Computation Tree Logic), a branching-time temporal logic;

Formal Specification
Temporal logics: timeline

One way of formally specifying properties of behaviours is using **temporal logics**.

A great deal of temporal logics have been proposed in the literature:

- **LTL** (Linear-time Temporal Logic) was introduced by Pnueli in 1977 [Pnu77];
- **CTL**, **CTL*** (Computation Tree Logic), a branching-time temporal logic;
- others (**CaReT** [AEM04], **HLTL**$^{\lightning}$, ...).

Formal Specification
LTL Syntax

LTL extends propositional logic with temporal modalities.

Formal Specification
LTL Syntax

LTL extends propositional logic with temporal modalities.

LTL syntax

LTL formulae over the set $\mathcal{AP}$ of atomic proposition are formed according to the following grammar:

$$\phi ::= \top \mid a \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \mathsf{X}\phi \mid \phi_1 \,\mathsf{U}\, \phi_2 \mid \mathsf{F}\phi \mid \mathsf{G}\phi$$

with $a \in \mathcal{AP}$.

Formal Specification
LTL Syntax

LTL extends propositional logic with temporal modalities.

LTL syntax

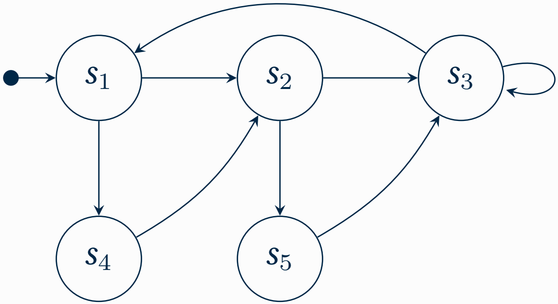
LTL formulae over the set $\mathcal{AP}$ of atomic proposition are formed according to the following grammar:

$$\phi ::= \top \mid a \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \mathsf{X}\phi \mid \phi_1\, \mathsf{U}\, \phi_2 \mid \mathsf{F}\phi \mid \mathsf{G}\phi$$

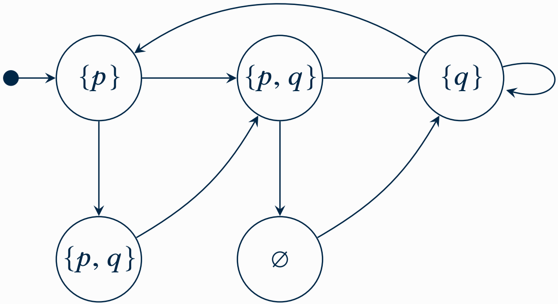
with $a \in \mathcal{AP}$.

LTL formulae are interpreted over system behaviours.

Formal Specification
From Transition Systems to Kripke Structures



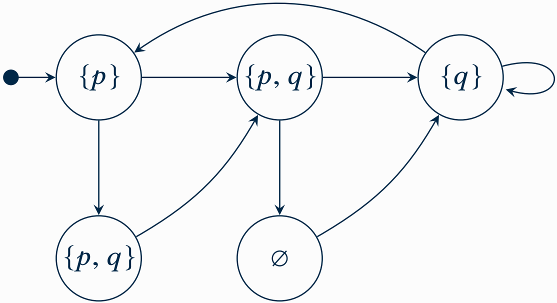
Formal Specification
From Transition Systems to Kripke Structures



- we associate a set of atomic propositions to each TS state;

Formal Specification

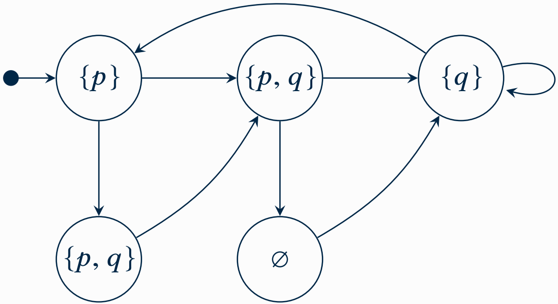
From Transition Systems to Kripke Structures



- we associate a set of atomic propositions to each TS state;
- a state s is labelled with the atomic proposition a iff a holds in s;

Formal Specification

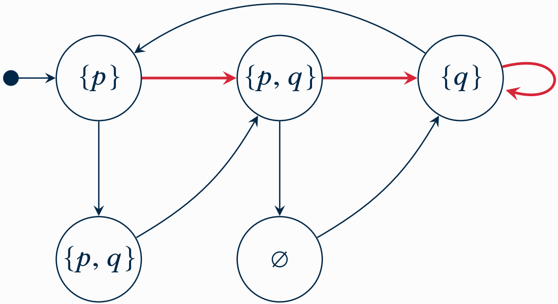
From Transition Systems to Kripke Structures



- we associate a set of atomic propositions to each TS state;
- a state *s* is labelled with the atomic proposition *a* iff *a* holds in *s*;
- in the above example, $\mathcal{AP} = \{p, q\}$;

Formal Specification

From Transition Systems to Kripke Structures



- we associate a set of atomic propositions to each TS state;
- a state s is labelled with the atomic proposition a iff a holds in s;
- in the above example, $\mathcal{AP} = \{p, q\}$;
- $\pi_1 = \{p\} \rightarrow \{p, q\} \rightarrow \{q\} \rightarrow \{q\} \rightarrow \{q\} \rightarrow \{q\} \rightarrow \{q\} \rightarrow \dots$

Formal Specification
LTL Semantics – Part 1

LTL syntax

$$\phi ::= \top \mid a \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \mathsf{X}\phi \mid \phi_1\,\mathsf{U}\,\phi_2 \mid \mathsf{F}\phi \mid \mathsf{G}\phi, \quad \text{with } a \in \mathcal{AP}.$$

Given a Kripke Structure behaviour $\pi = \pi_1 \rightarrow \pi_2 \rightarrow \ldots$, with $\pi_i \in \wp(\mathcal{AP})$, and LTL formula ϕ, the *satisfaction* relation $\pi \vDash \phi$ is defined inductively as follows:

Formal Specification
LTL Semantics – Part 1

LTL syntax

$$\phi ::= \top \mid a \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \mathsf{X}\phi \mid \phi_1\, \mathsf{U}\, \phi_2 \mid \mathsf{F}\phi \mid \mathsf{G}\phi, \quad \text{with } a \in \mathcal{AP}.$$

Given a Kripke Structure behaviour $\pi = \pi_1 \rightarrow \pi_2 \rightarrow \dots$, with $\pi_i \in \wp(\mathcal{AP})$, and LTL formula ϕ, the *satisfaction* relation $\pi \models \phi$ is defined inductively as follows:

- $\pi \models \top$;

Formal Specification
LTL Semantics – Part 1

LTL syntax

$$\phi ::= \top \mid a \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \mathsf{X}\phi \mid \phi_1\,\mathsf{U}\,\phi_2 \mid \mathsf{F}\phi \mid \mathsf{G}\phi, \quad \text{with } a \in \mathcal{AP}.$$

Given a Kripke Structure behaviour $\pi = \pi_1 \to \pi_2 \to \ldots$, with $\pi_i \in \wp(\mathcal{AP})$, and LTL formula ϕ, the *satisfaction* relation $\pi \models \phi$ is defined inductively as follows:

- $\pi \models \top$;
- $\pi \models a \in \mathcal{AP}$ iff $a \in \pi_1$;

Formal Specification

LTL Semantics – Part 1

LTL syntax

$$\phi ::= \top \mid a \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \mathsf{X}\phi \mid \phi_1 \,\mathsf{U}\, \phi_2 \mid \mathsf{F}\phi \mid \mathsf{G}\phi, \quad \text{with } a \in \mathcal{AP}.$$

Given a Kripke Structure behaviour $\pi = \pi_1 \to \pi_2 \to \ldots$, with $\pi_i \in \wp(\mathcal{AP})$, and LTL formula ϕ, the *satisfaction* relation $\pi \models \phi$ is defined inductively as follows:

- $\pi \models \top$;
- $\pi \models a \in \mathcal{AP}$ iff $a \in \pi_1$;
- $\pi \models \neg\phi$ iff $\pi \not\models \phi$;

Formal Specification
LTL Semantics – Part 1

LTL syntax

$$\phi ::= \top \mid a \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \mathsf{X}\phi \mid \phi_1 \,\mathsf{U}\, \phi_2 \mid \mathsf{F}\phi \mid \mathsf{G}\phi, \quad \text{with } a \in \mathcal{AP}.$$

Given a Kripke Structure behaviour $\pi = \pi_1 \rightarrow \pi_2 \rightarrow \ldots$, with $\pi_i \in \wp(\mathcal{AP})$, and LTL formula ϕ, the *satisfaction* relation $\pi \models \phi$ is defined inductively as follows:

- $\pi \models \top$;
- $\pi \models a \in \mathcal{AP}$ iff $a \in \pi_1$;
- $\pi \models \neg\phi$ iff $\pi \not\models \phi$;
- $\pi \models \phi_1 \wedge \phi_2$ iff $\pi \models \phi_1$ and $\pi \models \phi_2$;

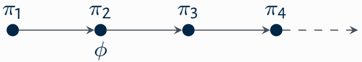
Formal Specification

LTL Semantics – Part 2

LTL syntax

$$\phi ::= \top \mid a \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \mathsf{X}\phi \mid \phi_1 \,\mathsf{U}\, \phi_2 \mid \mathsf{F}\phi \mid \mathsf{G}\phi, \quad \text{with } a \in \mathcal{AP}.$$

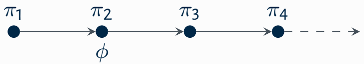
- $\pi \models \mathsf{X}\phi$ iff ϕ holds in the **next** moment in time;



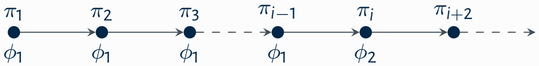
LTL syntax

$$\phi ::= \top \mid a \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \mathsf{X}\phi \mid \phi_1 \,\mathsf{U}\, \phi_2 \mid \mathsf{F}\phi \mid \mathsf{G}\phi, \quad \text{with } a \in \mathcal{AP}.$$

- $\pi \models \mathsf{X}\phi$ iff ϕ holds in the **next** moment in time;



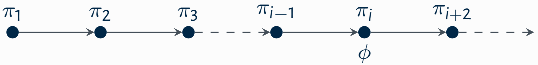
- $\pi \models \phi_1 \,\mathsf{U}\, \phi_2$ iff ϕ_2 holds in a future moment, and ϕ_1 is true **until** ϕ_2 holds;



LTL syntax

$$\phi ::= \top \mid a \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \mathsf{X}\phi \mid \phi_1 \,\mathsf{U}\, \phi_2 \mid \mathsf{F}\phi \mid \mathsf{G}\phi, \quad \text{with } a \in \mathcal{AP}.$$

- $\pi \models \mathsf{F}\phi$ iff ϕ **finally** holds sometime in the future;



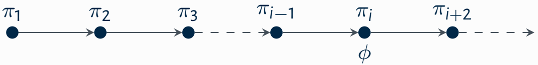
Formal Specification

LTL Semantics – Part 3

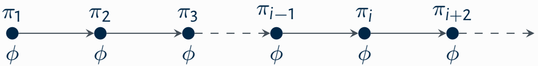
LTL syntax

$$\phi ::= \top \mid a \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \mathsf{X}\phi \mid \phi_1 \,\mathsf{U}\, \phi_2 \mid \mathsf{F}\phi \mid \mathsf{G}\phi, \quad \text{with } a \in \mathcal{AP}.$$

- $\pi \models \mathsf{F}\phi$ iff ϕ **finally** holds sometime in the future;



- $\pi \models \mathsf{G}\phi$ iff ϕ holds **globally** (now and in every future moment);



The LTL Model Checking problem

Given a Kripke Structure $\mathcal{M}$ and an LTL formula ϕ, we say that

$$\mathcal{M} \vDash \phi$$

iff $\pi \vDash \phi$, for each behaviour π of $\mathcal{M}$.

The LTL Model Checking problem

Given a Kripke Structure $\mathcal{M}$ and an LTL formula ϕ, we say that

$$\mathcal{M} \models \phi$$

iff $\pi \models \phi$, for each behaviour π of $\mathcal{M}$.

LTL Model Checking

The Model Checking problem amounts to decide whether $\mathcal{M} \models \phi$.

The LTL Model Checking problem
Some examples

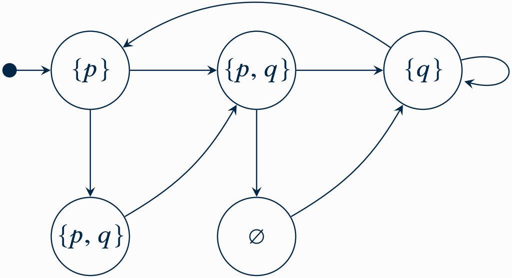


Figure: The Kripke Structure $\mathcal{M}$

The LTL Model Checking problem
Some examples

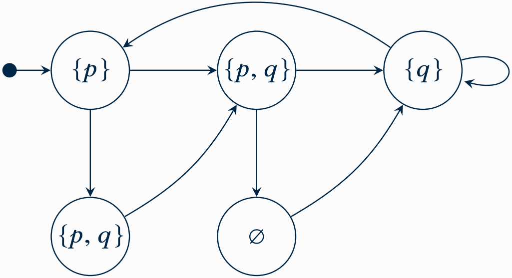


Figure: The Kripke Structure $\mathcal{M}$

$$\mathcal{M} \overset{?}{\models} q \vee \mathrm{X}q$$

The LTL Model Checking problem

Some examples

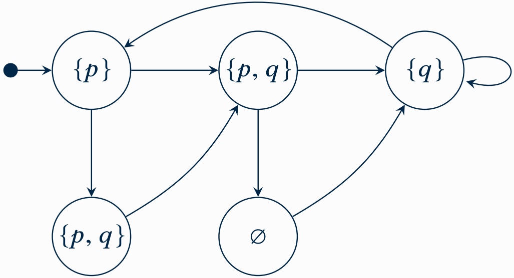


Figure: The Kripke Structure $\mathcal{M}$

$$\mathcal{M} \vDash q \vee \mathsf{X}q$$

The LTL Model Checking problem
Some examples

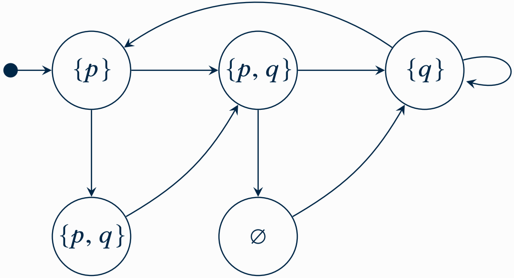


Figure: The Kripke Structure $\mathcal{M}$

$$\mathcal{M} \overset{?}{\models} \mathsf{G}(p \vee q)$$

The LTL Model Checking problem
Some examples

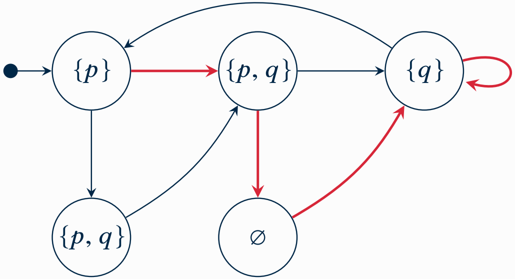


Figure: The Kripke Structure $\mathcal{M}$

$$\mathcal{M} \not\models \mathsf{G}(p \vee q)$$

The LTL Model Checking problem
Some examples

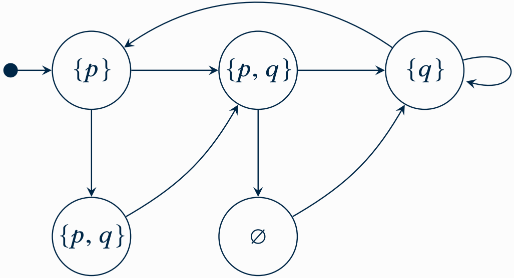


Figure: The Kripke Structure $\mathcal{M}$

$$\mathcal{M} \overset{?}{\models} \mathsf{G}((p \land q) \Rightarrow \mathsf{X}q)$$

The LTL Model Checking problem
Some examples

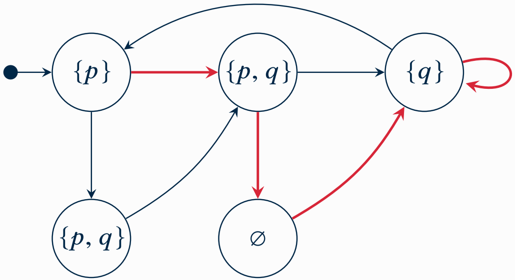


Figure: The Kripke Structure $\mathcal{M}$

$$\mathcal{M} \not\models \mathsf{G}((p \wedge q) \Rightarrow \mathsf{X}q)$$

The LTL Model Checking problem

Some examples

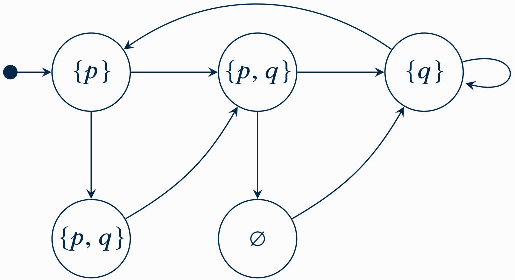


Figure: The Kripke Structure $\mathcal{M}$

$$\mathcal{M} \overset{?}{\models} \mathsf{GF}q$$

The LTL Model Checking problem
Some examples

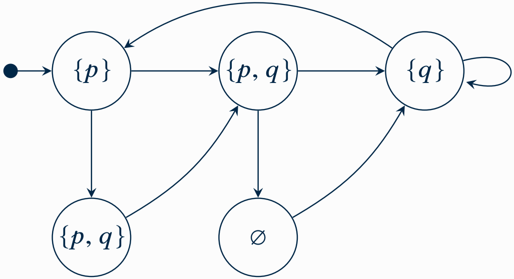


Figure: The Kripke Structure $\mathcal{M}$

$$\mathcal{M} \models \mathsf{GF}q$$

The LTL Model Checking problem
Some examples

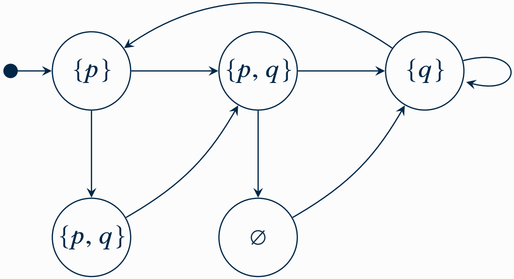


Figure: The Kripke Structure $\mathcal{M}$

$$\mathcal{M} \overset{?}{\models} \neg \mathsf{FG} q$$

THE LTL MODEL CHECKING PROBLEM
SOME EXAMPLES

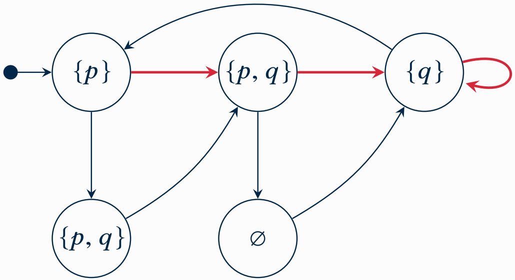
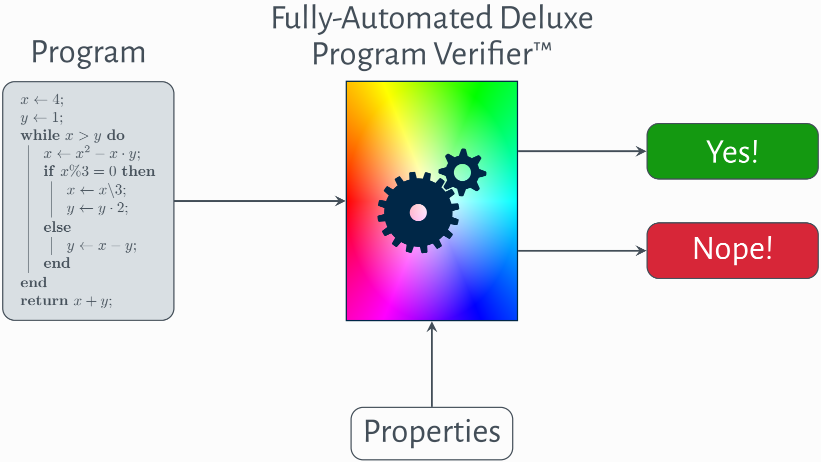


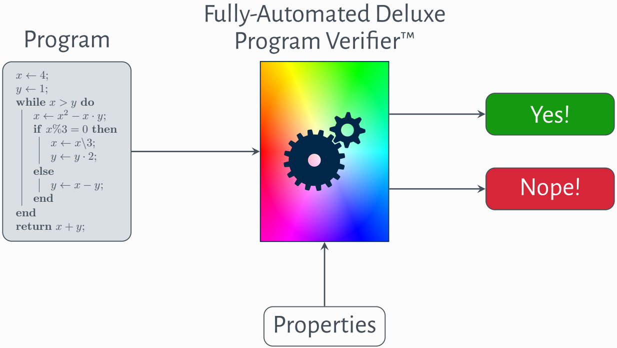
Figure: The Kripke Structure $\mathcal{M}$

$$\mathcal{M} \not\models \neg \mathsf{F}\mathsf{G} q$$

The dream of Automatic Verification

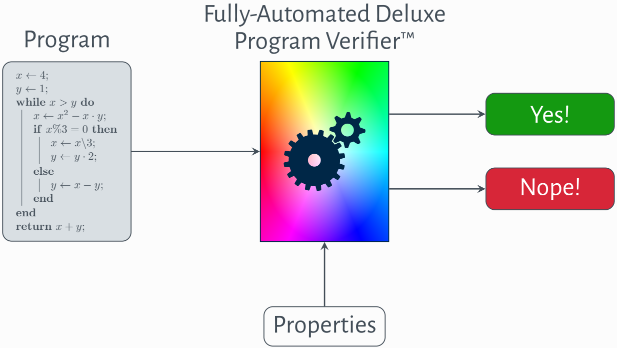


The dream of Automatic Verification
Achievable?



- we know that some properties of programs are undecidable, e.g. termination! (remember the halting problem?)

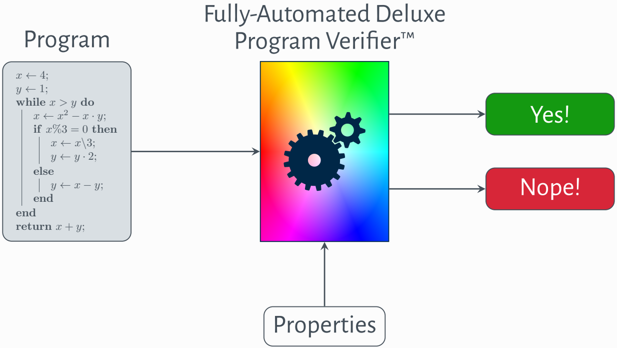
The dream of Automatic Verification
Achievable?



- we know that some properties of programs are undecidable, e.g. termination! (remember the halting problem?)
- perhaps other interesting properties are decidable?

The dream of Automatic Verification

Achievable?



- we know that some properties of programs are undecidable, e.g. termination! (remember the halting problem?)
- perhaps other interesting properties are decidable? Bad news...

The foundamental limit
Undecidability

Rice's theorem [RVG]

Every non-trivial semantic property of programs is undecidable.

- a property is non-trivial if it neither is true for every program nor it's false for every program;
- a semantic property is one about the program's behaviour.

The foundamental limit
Undecidability

Rice's theorem [RVG]

Every non-trivial semantic property of programs is undecidable.

- a property is non-trivial if it neither is true for every program nor it's false for every program;
- a semantic property is one about the program's behaviour.

An example

The property of returning 0 for some input is undecidable by Rice's Theorem.

The foundamental limit
Undecidability

Implicit in Rice's Theorem is an idealized program model.

- Turing Machines have unbounded memory;
- A variable in Martin Davis' $\mathcal{S}$ programs can be incremented indefinitely and never overflows;

The foundamental limit
Undecidability

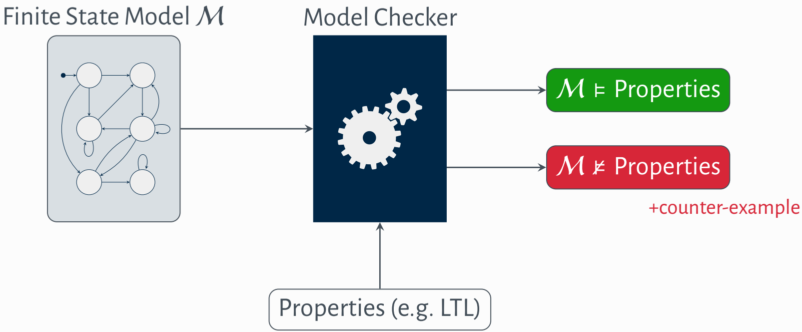
Implicit in Rice's Theorem is an idealized program model.

- Turing Machines have unbounded memory;
- A variable in Martin Davis' $\mathcal{S}$ programs can be incremented indefinitely and never overflows;

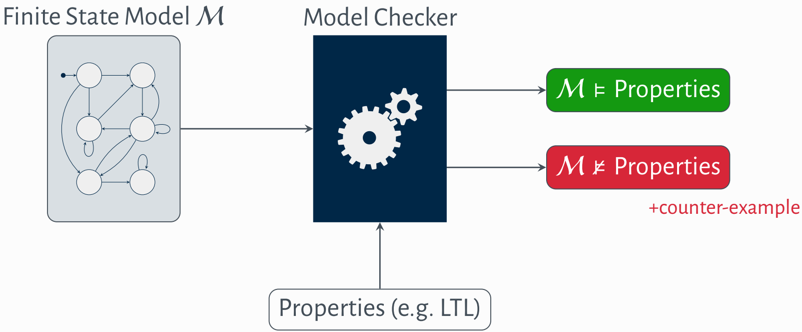
Concrete computing devices have **bounded** resources!

The model checking problem is decidable if we restrict ourselves to finite-state models.

Automatic Verification
Model Checkers



Automatic Verification
Model Checkers



Some well-known model checkers are [SPIN], [nuSMV], [TLC], [JPF].

The practical limit
State space explosion

- A finite state space can always be generated and explored in finite time.

The practical limit
State space explosion

- A finite state space can always be generated and explored in finite time.
- Unfortunately, this does not mean that doing so is always feasible, as the state space can get very large!

The practical limit

State space explosion

- A finite state space can always be generated and explored in finite time.
- Unfortunately, this does not mean that doing so is always feasible, as the state space can get very large!
- 1KB of memory (1 000 B) yields $2^{8000} \approx 10^{2408}$ states;

The practical limit
State space explosion

- A finite state space can always be generated and explored in finite time.
- Unfortunately, this does not mean that doing so is always feasible, as the state space can get very large!
- 1KB of memory (1 000 B) yields $2^{8000} \approx 10^{2408}$ states;
- 10 `double` variables (64 bit each) yield $2^{10\times 64} \approx 10^{192}$ states;

The practical limit
State space explosion

- A finite state space can always be generated and explored in finite time.
- Unfortunately, this does not mean that doing so is always feasible, as the state space can get very large!
- 1KB of memory (1 000 B) yields $2^{8000} \approx 10^{2408}$ states;
- 10 `double` variables (64 bit each) yield $2^{10\times 64} \approx 10^{192}$ states;
- optimistic limit for a model checker? 10^{100} states [Kwo00].

Formal Methods in Software Engineering

Using Formal Methods (FM)

- FM can be used along with traditional development methodologies.

Using Formal Methods (FM)

- FM can be used along with traditional development methodologies.
- During Analysis and Design, FM can:
  - be a solid foundation for describing complex systems;
  - help with early detection of faults.

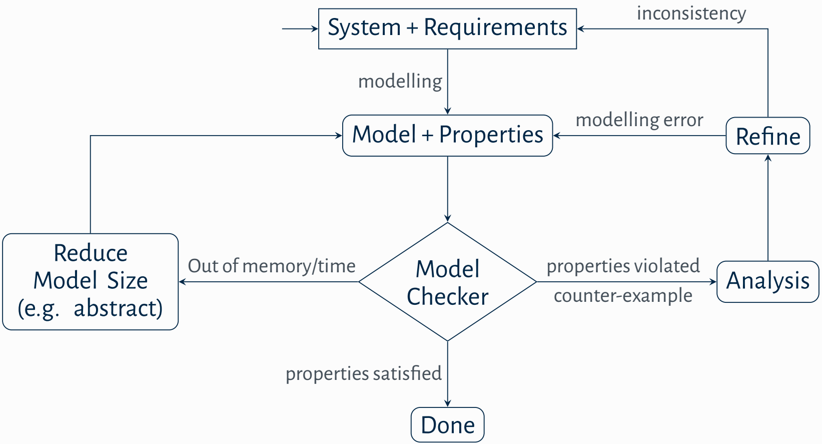
Using Formal Methods (FM)

- FM can be used along with traditional development methodologies.
- During Analysis and Design, FM can:
  - be a solid foundation for describing complex systems;
  - help with early detection of faults.
- During Development, FM can:
  - provide support with synthesis techniques.

Using Formal Methods (FM)

- FM can be used along with traditional development methodologies.
- During Analysis and Design, FM can:
  - be a solid foundation for describing complex systems;
  - help with early detection of faults.
- During Development, FM can:
  - provide support with synthesis techniques.
- During Verification, FM can:
  - increase the confidence on system reliability;
  - help with traditional verification techniques (e.g. test case generation).

The Model Checking process



A success story
Formal Methods at Amazon Web Services – part 1

- Software Engineers at AWS use Formal Methods [New+14];

A success story
Formal Methods at Amazon Web Services – part 1

- Software Engineers at AWS use Formal Methods [New+14];
- To verify correctness of DynamoDB production code:

A success story
Formal Methods at Amazon Web Services – part 1

- Software Engineers at AWS use Formal Methods [New+14];
- To verify correctness of DynamoDB production code:
  - extensive fault-injection testing using a simulated network layer to control message loss, duplication, and re-ordering;

A success story
Formal Methods at Amazon Web Services – part 1

- Software Engineers at AWS use Formal Methods [New+14];
- To verify correctness of DynamoDB production code:
  - extensive fault-injection testing using a simulated network layer to control message loss, duplication, and re-ordering;
  - stress tests for long periods on real hardware under many different workloads;

A success story
Formal Methods at Amazon Web Services – part 1

- Software Engineers at AWS use Formal Methods [New+14];
- To verify correctness of DynamoDB production code:
  - extensive fault-injection testing using a simulated network layer to control message loss, duplication, and re-ordering;
  - stress tests for long periods on real hardware under many different workloads;
  - detailed informal proofs of correctness (found several bugs);

A success story
Formal Methods at Amazon Web Services – part 1

- Software Engineers at AWS use Formal Methods [New+14];
- To verify correctness of DynamoDB production code:
  - extensive fault-injection testing using a simulated network layer to control message loss, duplication, and re-ordering;
  - stress tests for long periods on real hardware under many different workloads;
  - detailed informal proofs of correctness (found several bugs);
  - Formal Methods and Model Checking (using TLC).

A success story
Formal Methods at Amazon Web Services – part 2

- In two week, they learned how to use TLA+/TLC and wrote a detailed specification;

A success story

Formal Methods at Amazon Web Services – part 2

- In two week, they learned how to use TLA+/TLC and wrote a detailed specification;
- Model-checked the specification using 10 EC2 instances, each with 8 cores plus hyperthreads, and 23 GB of RAM;

A success story

Formal Methods at Amazon Web Services – part 2

- In two week, they learned how to use TLA+/TLC and wrote a detailed specification;
- Model-checked the specification using 10 EC2 instances, each with 8 cores plus hyperthreads, and 23 GB of RAM;
- Found a data-loss bug if a particular sequence of failures and recovery steps was interleaved with other processing; the shortest error trace exhibiting the bug contained 35 high-level steps.

A success story

Formal Methods at Amazon Web Services – part 3

- This success led to management advocating TLA+ to other teams working on other products;

Product	Component	Benefits
DynamoDB	Replication & group-membership system	Found 3 bugs.
S3	Fault-tolerant low-level network algorithm	Found 2 bugs. Found further bugs in proposed optimizations.
	Background redistribution of data	Found 1 bug, and found a bug in the first proposed fix.
EBS	Volume management	Found 3 bugs.

Table: Benefits of using Formal Methods on different products at AWS

Model Checking: weaknesses

- Limits: may be undecidable or unfeasible (state space explosion);

Model Checking: weaknesses

- Limits: may be undecidable or unfeasible (state space explosion);
- It verifies a *model*, and not the actual system itself; the results are only as good as the model.

Model Checking: weaknesses

- Limits: may be undecidable or unfeasible (state space explosion);
- It verifies a *model*, and not the actual system itself; the results are only as good as the model.
- Requires expertise in finding adequate abstractions and stating properties;

Model Checking: weaknesses

- Limits: may be undecidable or unfeasible (state space explosion);
- It verifies a *model*, and not the actual system itself; the results are only as good as the model.
- Requires expertise in finding adequate abstractions and stating properties;
- As with any tool, a model checker may contain software defects!

Model Checking: strenghts

- Can provide a significant increase in the level of confidence of system correctness;

Model Checking: strenghts

- Can provide a significant increase in the level of confidence of system correctness;
- It is a potential "push-button" technology;

Model Checking: strenghts

- Can provide a significant increase in the level of confidence of system correctness;
- It is a potential "push-button" technology;
- It can be easily integrated in existing development methodologies;

Model Checking: strenghts

- Can provide a significant increase in the level of confidence of system correctness;
- It is a potential "push-button" technology;
- It can be easily integrated in existing development methodologies;
- It provides useful diagnostic counter-examples in case a property is violated;

Practice time!

A concurrent program


```
process P0 {
  while(true){
    // noncritical section
    flag_0 = 1;
    while (flag_1) {}
    // critical section
    flag_0 = 0;
    // noncritical section
  }
}
```

```
process P1 {
  while(true){
    // noncritical section
    flag_1 = 1;
    while (flag_0) {}
    // critical section
    flag_1 = 0;
    // noncritical section
  }
}
```

A concurrent program

Modelling


```
process P0 {
  while(true){
    // noncritical section
    flag_0 = 1;
    while (flag_1) {}
    // critical section
    flag_0 = 0;
    // noncritical section
  }
}
```

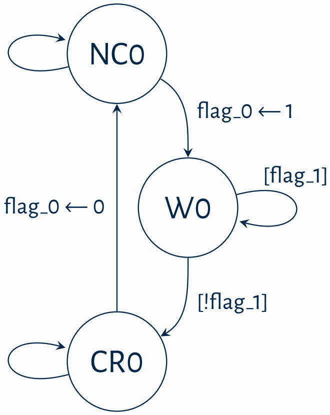


Figure: Model for process P0

A concurrent program

Modelling

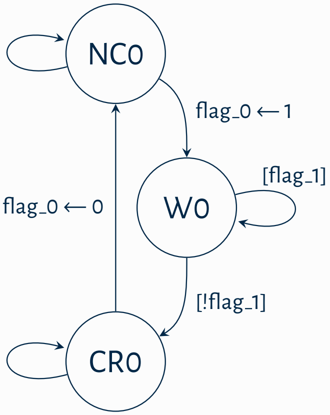


Figure: Model for process `P0`

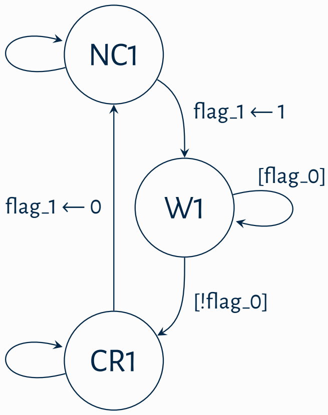


Figure: Model for process `P1`

A concurrent program
Modelling: parallel composition

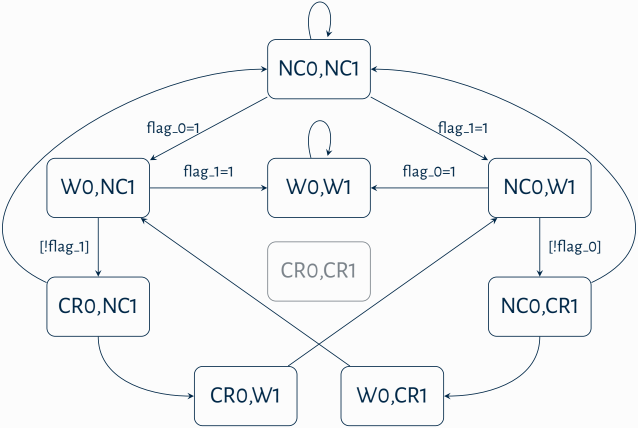


Figure: Asynchronous parallel composition of `P0` and `P1`

Demo time
Model Checking with Spin/Promela

Take Home Messages

Take-home Messages

- Traditional verification techniques (and their limits);

Take-home Messages

- Traditional verification techniques (and their limits);
- Formal Methods

Take-home Messages

- Traditional verification techniques (and their limits);
- Formal Methods
  - System Specification (Transition Systems, higher-level specification languages);

Take-home Messages

- Traditional verification techniques (and their limits);
- Formal Methods
  - System Specification (Transition Systems, higher-level specification languages);
  - Property Specification (LTL);

Take-home Messages

- Traditional verification techniques (and their limits);
- Formal Methods
  - System Specification (Transition Systems, higher-level specification languages);
  - Property Specification (LTL);
  - System Verification (Model Checking);

Take-home Messages

- Traditional verification techniques (and their limits);
- Formal Methods
  - System Specification (Transition Systems, higher-level specification languages);
  - Property Specification (LTL);
  - System Verification (Model Checking);
- Using Formal Methods;

Any questions?

References I

[AEM04] Rajeev Alur, Kousha Etessami, and Parthasarathy Madhusudan. "A temporal logic of nested calls and returns". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2004, pp. 467–481.

[AKY99] Rajeev Alur, Sampath Kannan, and Mihalis Yannakakis. "Communicating hierarchical state machines". In: *International Colloquium on Automata, Languages, and Programming*. Springer. 1999, pp. 169–178.

[Ben+17] Massimo Benerecetti et al. "Dynamic state machines for modelling railway control systems". In: *Science of Computer Programming* 133 (2017), pp. 116–153.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. Vol. 26202649. Jan. 2008. ISBN: 978-0-262-02649-9.

References II

[Dij72] E. W. Dijkstra. "The humble programmer [1972 ACM Turing Award Lecture]". In: *Communications of the ACM* 15.10 (1972), pp. 859–866.

[Gli] Martin Glinz. "Statecharts for requirements specification-as simple as possible, as rich as needed". In:

[Har87] David Harel. "Statecharts: A visual formalism for complex systems". In: *Science of computer programming* 8.3 (1987), pp. 231–274.

[HN96] David Harel and Amnon Naamad. "The STATEMATE semantics of statecharts". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 5.4 (1996), pp. 293–333.

[JPF] *JPF - Java PathFinder*. URL: http://javapathfinder.sourceforge.net/.

References III

[Kwo00] Gihwon Kwon. “Rewrite Rules and Operational Semantics for Model Checking UML Statecharts”. In: *<UML> 2000 — The Unified Modeling Language*. Ed. by Andy Evans, Stuart Kent, and Bran Selic. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 528–540. ISBN: 978-3-540-40011-0.

[LO82] Leslie Lamport and Susan Owicki. “Proving liveness properties of concurrent programs”. In: *ACM Transactions on Programming Languages and Systems* 4 (1982).

[New+14] Chris Newcombe et al. “Use of formal methods at Amazon Web Services”. In: (2014).

[nuSMV] *nuSMV home page*. URL: http://nusmv.fbk.eu/.

[Pnu77] A. Pnueli. “The temporal logic of programs”. In: *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*. Oct. 1977, pp. 46–57. DOI: 10.1109/SFCS.1977.32.

REFERENCES IV

[PRO] *Promela Language Reference*. URL: http://spinroot.com/spin/Man/promela.html (visited on 05/05/2019).

[RVG] Rob van Glabbeek. *Rice's theorem*. URL: http://kilby.stanford.edu/~rvg/154/handouts/Rice.html.

[SPIN] *SPIN - Formal Verification*. URL: http://spinroot.com/spin/whatispin.html.

[TLC] Leslie Lamport. *The TLA+ Home Page*. URL: http://lamport.azurewebsites.net/tla/tla.html.