

Department of Mathematics, Computer Science and Physics, University of Udine

The Safety Fragment of Temporal Logics on Infinite Sequences

Lesson 8

Luca Geatti

luca.geatti@uniud.it

Angelo Montanari

angelo.montanari@uniud.it

April 29th, 2024



- ① Background
 - ① Regular and ω -regular languages
 - ② The First- and Second-order Theory of One Successor
 - ③ Automata over finite and infinite words
 - ④ Linear Temporal Logic
- ② The safety fragment of LTL and its theoretical features
 - ① Definition of Safety and Cosafety
 - ② Characterizations and Normal Forms
 - ③ Kupferman and Vardi's Classification



- ③ Recognizing safety
 - ① Recognizing safety Büchi automata
 - ② Recognizing safety formulas of LTL
 - ③ Construction of the automaton for the bad prefixes
- ④ Algorithms and Complexity
 - ① Satisfiability
 - ② Model Checking
 - ③ Reactive Synthesis
- ⑤ Succinctness and Pastification
 - ① Succinctness of Safety Fragments
 - ② Pastification Algorithms

ALGORITHMS & COMPLEXITY

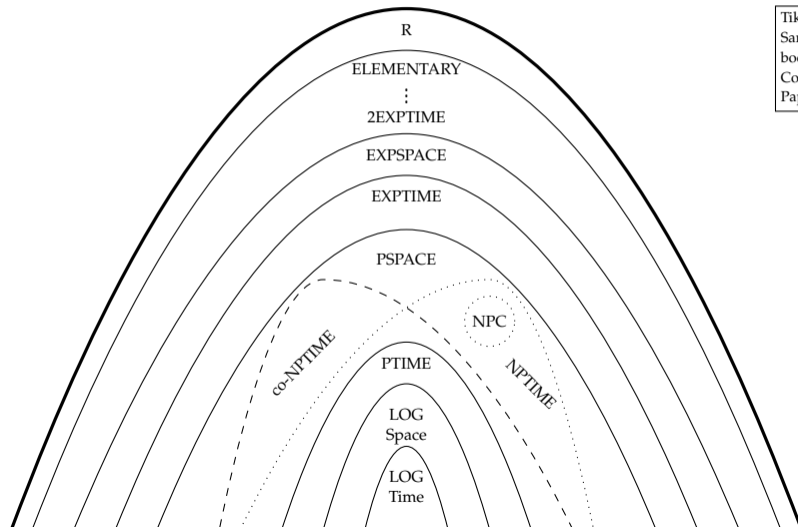
for the safety fragment of LTL



- Efficient algorithms and theoretical complexity for the problems of:
 - satisfiability
 - model checking
 - symbolic algorithms
 - exploiting the Kupferman & Vardi's classification (informative prefixes)
 - reactive synthesis



Recap of complexity classes



TikZ code by Sebastian Sardiña, based on the book "Computational Complexity" by C. H. Papadimitriou

SATISFIABILITY

of (co)safety fragments of LTL



The satisfiability problem

Let \mathbb{L} be a temporal logic over infinite sequences.

Definition

Given a formula ϕ of \mathbb{L} , we say that ϕ is satisfiable iff $\mathcal{L}(\phi) \neq \emptyset$.

The **satisfiability problem** of \mathbb{L} is the problem of checking whether a given input formula ϕ is satisfiable.



The satisfiability problem of LTL (LTL-SAT) is PSPACE-complete.

- same for LTL+P

Reference:

A Prasad Sistla and Edmund M Clarke (1985). "The complexity of propositional linear temporal logics".
In: *Journal of the ACM (JACM)* 32.3, pp. 733–749. DOI: 10.1145/3828.3837

Classic Algorithm

Given $\phi \in \text{LTL}+P$ of size n ,

- build an NBA \mathcal{A} such that:
 - $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\phi)$
 - $|\mathcal{A}| \in 2^{\mathcal{O}(n)}$
- check the (non)emptiness of \mathcal{A} , i.e., the existence of a *final state* q such that:
 - $q_0 \rightsquigarrow q$
 - $q \rightsquigarrow q$



The satisfiability problem of LTL (LTL-SAT) is PSPACE-complete.

- same for LTL+P

Reference:

A Prasad Sistla and Edmund M Clarke (1985). "The complexity of propositional linear temporal logics".
In: *Journal of the ACM (JACM)* 32.3, pp. 733–749. DOI: 10.1145/3828.3837

Classic Algorithm

Complexity:

- nonemptiness = reachability, thus nondeterministic logarithmic space
- can be done *on-the-fly* while building the NBA
- Total: nondeterministic polynomial space (PSPACE)



The satisfiability problem of LTL_f (**LTL_f-SAT**) is PSPACE-complete.

Reference:

Giuseppe De Giacomo and Moshe Y. Vardi (2013). "Linear Temporal Logic and Linear Dynamic Logic on Finite Traces". In: *Proceedings of the 23rd International Joint Conference on Artificial Intelligence*. Ed. by Francesca Rossi. IJCAI/AAAI, pp. 854–860

Classic Algorithm

Complexity:

- nonemptiness = reachability, thus nondeterministic logarithmic space
- can be done **on-the-fly** while building the NBA
- Total: nondeterministic polynomial space (PSPACE)



Theorem

The satisfiability problem for the logics SafetyLTL, G(pLTL), coSafetyLTL, F(pLTL), and LTL[X, G] is PSPACE-complete.

Reference:

Alessandro Artale et al. (2023). “Complexity of Safety and coSafety Fragments of Linear Temporal Logic”. In: *Proc. of the 36th AAAI Conf. on Artificial Intelligence*. AAAI Press



Theorem

The satisfiability problem for the logics SafetyLTL, G(pLTL), coSafetyLTL, F(pLTL), and LTL[X, G] is PSPACE-complete.

Proof.

- membership: from LTL-SAT
- hardness: reduction from LTL_f -SAT





Theorem

The satisfiability problem for the logics SafetyLTL, G(pLTL), coSafetyLTL, F(pLTL), and LTL[X, G] is PSPACE-complete.

The restriction to (co)safety fragments, i.e., the restriction on reasoning over finite traces, does not change the worst-case complexity of the satisfiability problem.



Theorem

The satisfiability problem for $LTL[X, F]$ is NP-complete.

Lemma (Small model property)

For any $\phi \in LTL[X, F]$, it holds that ϕ is satisfiable iff there exists a trace σ such that:

- $\sigma \models \phi$
- $|\sigma| \leq |\phi|$



Theorem

The satisfiability problem for $LTL[X, F]$ is NP-complete.

Proof.

- membership: **nondeterministic** algorithm
 - guess an $n \leq |\phi|$ and the assignments for the first n states of a candidate trace σ
 - check whether $\sigma \cdot (2^{\mathcal{AP}})^\omega \models \phi$
 - if at least one candidate model is indeed a correct model, terminate with **SAT**; otherwise terminate with **UNSAT**.
- hardness: from Boolean satisfiability





Theorem

The satisfiability problem for $LTL[X, F]$ is NP-complete.

Reference:

A Prasad Sistla and Edmund M Clarke (1985). “The complexity of propositional linear temporal logics”. In: *Journal of the ACM (JACM)* 32.3, pp. 733–749. DOI: 10.1145/3828.3837



Logics	Problems		
	satisfiability	model checking	realizability
coSafetyLTL	PSPACE-c	???	2EXPTIME-c
F(pLTL)	PSPACE-c	???	EXPTIME-c
LTL[X, F]	NP-c	???	EXPTIME-c

Logics	Problems		
	satisfiability	model checking	realizability
SafetyLTL	PSPACE-c	???	2EXPTIME-c
G(pLTL)	PSPACE-c	???	EXPTIME-c
LTL[\tilde{X} , G]	PSPACE-c	???	EXPTIME-c

MODEL CHECKING

an introduction

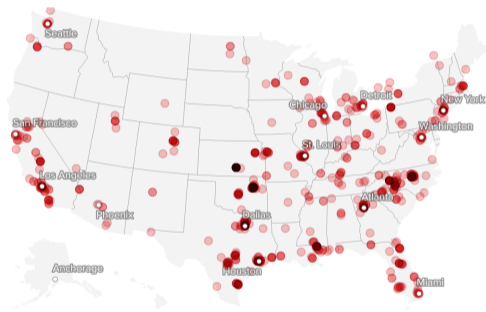
based on the slides of Prof. Joost-Pieter Katoen



- Therac-25 was a medical linear accelerator used for radiation therapy in the 1980s.
- It caused six known accidents between 1985 and 1987 due to software and hardware errors (*race condition*).
- Patients received massive overdoses of radiation, leading to serious injuries and fatalities.
- The accidents were attributed to software bugs and a lack of proper safety mechanisms.

AT&T outages and reported problems

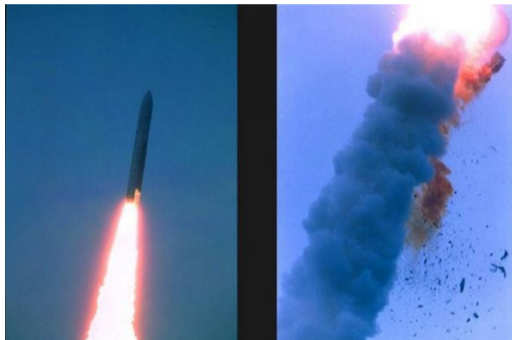
The map shows the concentration of user-submitted problem reports over the last 24 hours to Downdetector. The darker red areas show higher concentrations of reports.



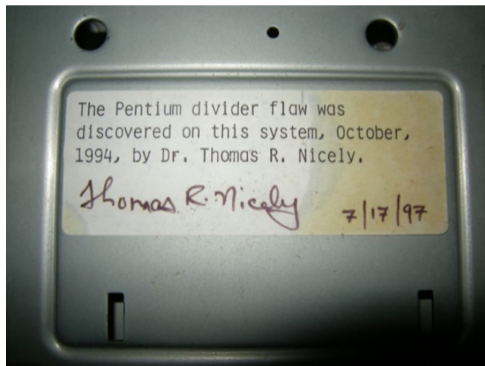
Note: Data as of 10 a.m. EST on Feb. 22, 2024.

Map: Taylor Johnston / CBS News • Source: Downdetector

- In 1990, AT&T experienced a widespread network outage affecting millions of customers.
- The outage was caused by a software bug triggered during a routine maintenance procedure.
 - wrong interpretation of break statement in C
- The bug led to the failure of crucial network components, disrupting telephone services across the United States.



- Crash of the european Ariane 5-missile in June 1996
- Costs: more than 500 million USD
- Source: software flaw in the control software. A data conversion from a 64-bit floating point to 16-bit signed integer.
- Efficiency considerations had led to the disabling of the software handler (in Ada)



- Pentium FDIV bug: a significant hardware flaw in Intel's Pentium microprocessor in 1994.
- Incorrect calculations in floating-point division operations
- The bug affected a small percentage of Pentium processors but garnered widespread attention.
- Intel initially downplayed the issue but eventually offered free replacements for affected chips, resulting in a costly recall program and damage to the company's reputation (500M USD).



- Speech@50-years Celebration CWI Amsterdam
- *"It is fair to state, that in this digital era correct systems for information processing are more valuable than gold."*
- Henk Barendregt



The Importance of Software Correctness

Rapidly increasing integration of ICT in different applications

- embedded systems
- communication protocols
- transportation systems
- ...

Defects can be fatal and extremely costly

- products subject to mass-production
- safety-critical systems



What is Software and Hardware Verification?

Verification:

- Verification answers the question: *"Are we building the product right?"*
- It ensures that the software product meets the specified requirements and adheres to the design and development specifications.
- It involves checking whether the software is being built correctly.
- Verification activities typically include reviews, inspections to identify defects early in the development process, and model checking.

Validation:

- Validation answers the question: *"Are we building the right product?"*
- It confirms that the software meets the needs and expectations of the end-users and stakeholders.
- It involves evaluating the software against user requirements and ensuring that it solves the intended problem effectively.
- Validation activities typically include testing the software against user scenarios, user acceptance testing (UAT), and usability testing.



Some examples of Verification Techniques

Reviews and Inspections:

- It involves a group of individuals examining software documentation, code, or design to identify defects, inconsistencies, or areas for improvement (*code reviews, design reviews, and walkthroughs*).

Static Analysis:

- analyze software code or documentation *without executing it*, aiming to identify potential issues such as syntax errors, coding standards violations, or security vulnerabilities.

Unit Testing:

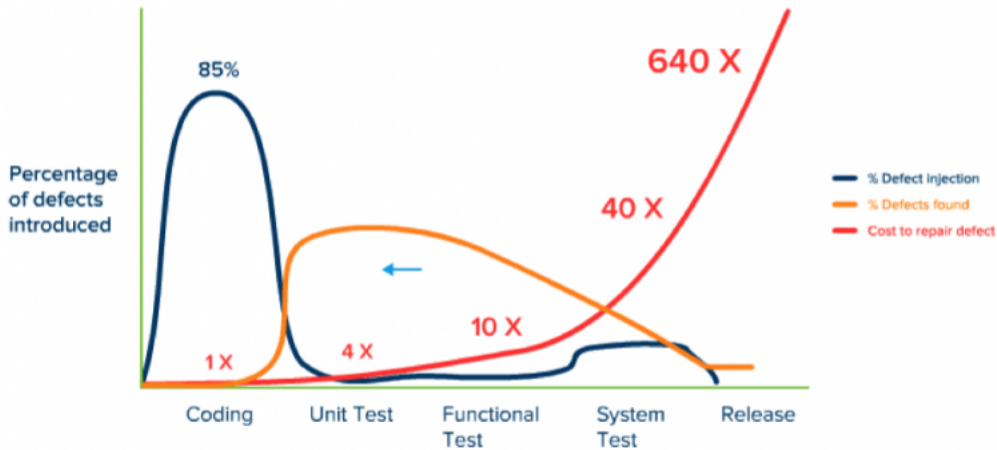
- Unit testing involves testing individual components or units of code in isolation to ensure they perform as expected. Developers write test cases to verify the behavior of specific functions or modules (*JUnit, pytest*).

Integration Testing:

- Verifies the interactions between different modules or components of a software system. It ensures that the integrated units work together seamlessly as intended.



Testing: the Sooner, the Better



Jones, Capers. *Applied Software Measurement: Global Analysis of Productivity and Quality*.



Definition (Formal Methods)

Formal methods are a set of mathematical techniques used to rigorously specify, model, and verify software and hardware systems. They involve the use of precise mathematical languages and logical reasoning to ensure correctness and reliability in the design and implementation of complex systems.

Advantages:

- early integration of verification in the design process
- providing more effective verification techniques (higher coverage)
- reducing the verification time (automatic techniques)

The usage of formal methods is highly recommended by IEC, FAA, and NASA for safety-critical software



3 categories of Formal Methods

Deductive Methods

- *method*: provide a formal proof that P holds
- *tool*: theorem prover / proof assistant or proof checker
- *applicable if*: system has form of a mathematical theory

Model Checking

- *method*: systematic check on P in all states
- *tool*: model checker (Spin, NuSMV, UppAal, ...)
- *applicable if*: system generates (finite) behavioural model

Model-based simulation or testing

- *method*: test for P by exploring possible behaviours
- *tool*: ...
- *applicable if*: system defines an executable model



Procedure:

- take a model (simulation) or a realisation (testing)
- stimulate it with certain inputs, i.e., the tests
- observe reaction and check whether this is “desired”

Important drawbacks:

- number of possible behaviours is very large (or even infinite)
- unexplored behaviours may contain the fatal bug
- testing/simulation can show the presence of errors, not their absence



$$\{p\} \text{ skip } \{p\}$$

$$\{p[x \mapsto t]\} x := t \{p\} \quad (2.54)$$

$$\frac{\{p\} S_1 \{r\} \quad \{r\} S_2 \{q\}}{\{p\} S_1; S_2 \{q\}}$$

$$\frac{\{p \wedge b\} S_1 \{q\} \quad \{p \wedge \neg b\} S_2 \{q\}}{\{p\} \text{ if } b \text{ then } S_1 \text{ else } S_2 \{q\}}$$

$$\frac{\{p \wedge b\} S \{p\}}{\{p\} \text{ while } b \text{ do } S \{p \wedge \neg b\}} \quad (2.55)$$

$$\frac{\{p_1\} S \{q_1\}}{\{p\} S \{q\}} \quad \text{provided } p \Rightarrow p_1 \text{ and } q_1 \Rightarrow q.$$

Figure 2.1: Hoare Logic for Partial Correctness – Syntactic Presentation



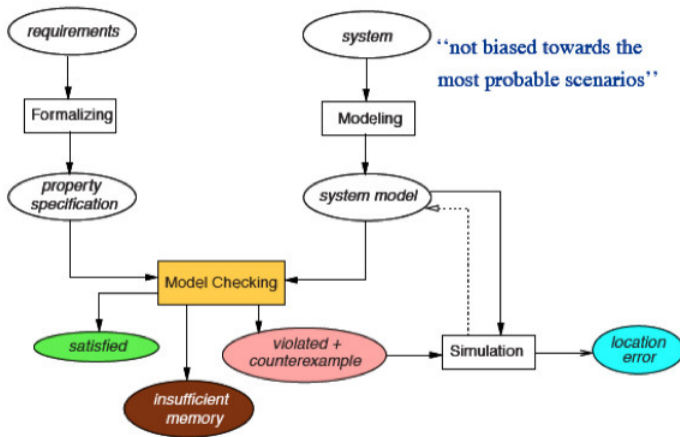
- 1949 (Turing) **Mathematical program correctness.** *An early program proof by Alan Turing – FL Morris, CB Jones (1984)*
- 1969 (Hoare) **Syntax-based technique for sequential programs.** For a given input, does a computer program generate the correct output? Based on compositional proof rules expressed in predicate logic
- 1977 (Pnueli) **Syntax-based technique for concurrent programs.** Handles properties referring to states during the computation. Based on proof rules expressed in temporal logic.
- 1980 (Clarke, Emerson, ...) **Automated verification of concurrent programs.** Model-based instead of proof-rule based approach. Does the concurrent program satisfy a given (logical) property?



The most used formal verification technique is **Model Checking** (**MC**, for short).

Distinctive features:

- fully automatic;
- exhaustive;
- it generates a counterexample trace if the specification does not hold.





(a) E.M. Clarke



(b) E.A. Emerson



(c) J. Sifakis

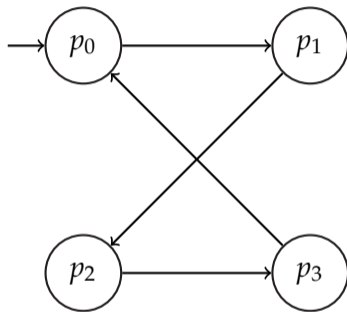
For their role in developing Model-Checking into a highly effective verification technology that is widely adopted in the hardware and software industries.



Definition (Kripke structure)

A Kripke structure (or **Transition System**) is a tuple $M = \langle \mathcal{AP}, Q, I, T, L \rangle$ where:

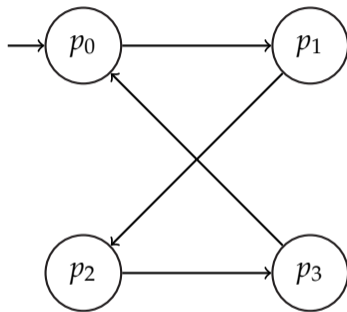
- \mathcal{AP} is a finite alphabet,
- Q is the finite set of states,
- $I \subseteq Q$ is the set of initial states,
- $T \subseteq Q \times Q$ is a *complete* transition relation, and
- $L : Q \rightarrow 2^{\mathcal{AP}}$ is the labeling function that assigns to each state the set of atoms in \mathcal{AP} that are true in that state.





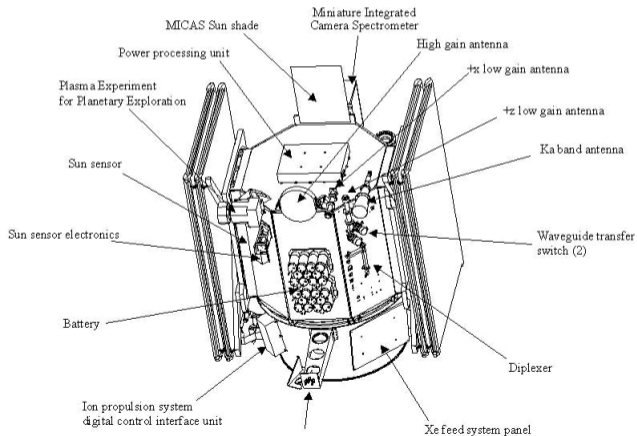
Expressivity:

- Programs are transition systems
- Multi-threading programs are transition systems
- Communicating processes are transition systems
- Hardware circuits are transition systems





NASA's Deep Space-1 Spacecraft (model checking has been applied to the modules of this spacecraft)





Example of properties

- Can the system reach a deadlock situation?
- Can two processes ever be simultaneously in a critical section?
- On termination, does a program provide the correct output?

Temporal Logics

- Linear Temporal Logic (LTL) and its variants
 - real-time LTL
 - Signal Temporal Logic
 - Linear Temporal Logic modulo theories
- Computation Tree Logic (CTL)

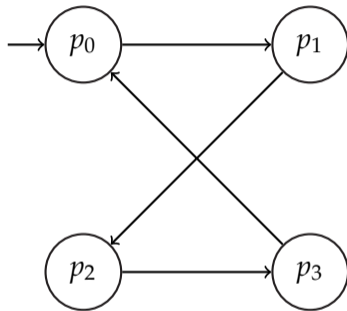
Definition (Model Checking of LTL)

Given:

- a Kripke structure
 $M = \langle \mathcal{AP}, Q, I, T, L \rangle$
- an initial state $s \in I$ of M
- an LTL formula ϕ over the set of atomic propositions \mathcal{AP}

we write $M, s \models A\phi$ iff all paths of M starting from s are models of ϕ .

A is the “for all paths” operator of CTL.

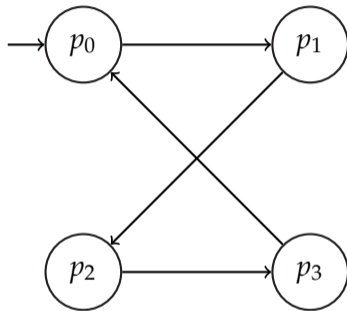


Definition (Model Checking of LTL)

The model checking problem of LTL (**LTL-MC**) is the problem of establishing whether $M, s \models A\phi$.

Example:

- $M, s \models GF(p_0)$
- $M, s \not\models FG(p_0)$



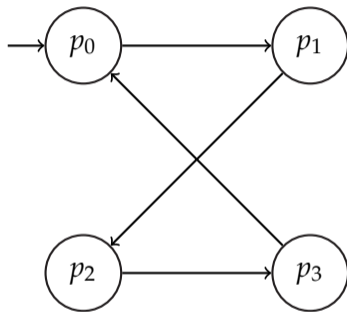


Theorem

The LTL-MC is PSPACE-complete.

Reference:

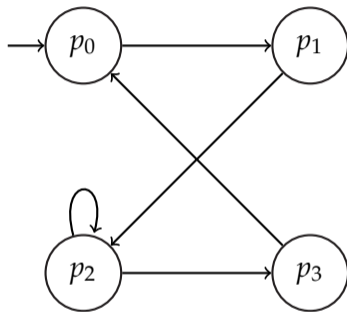
A Prasad Sistla and Edmund M Clarke (1985). "The complexity of propositional linear temporal logics".
In: *Journal of the ACM (JACM)* 32.3, pp. 733–749. DOI: 10.1145/3828.3837





Examples:

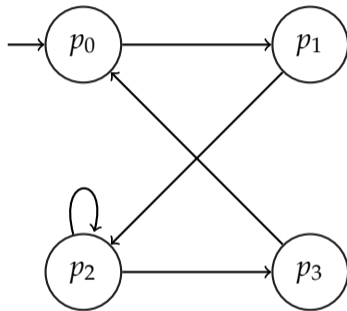
- $G(\bigwedge_{i=0}^3 (p_i \rightarrow \bigvee_{\substack{j=0 \\ j \neq i}}^3 p_j))$





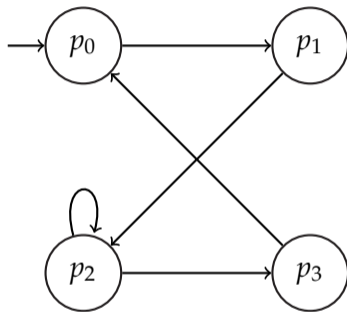
Examples:

- $G(\bigwedge_{i=0}^3 (p_i \rightarrow \bigvee_{\substack{j=0 \\ j \neq i}}^3 p_j))$
 - False
 - Counterexample: $\langle p_0, p_1, p_2, p_2 \rangle$



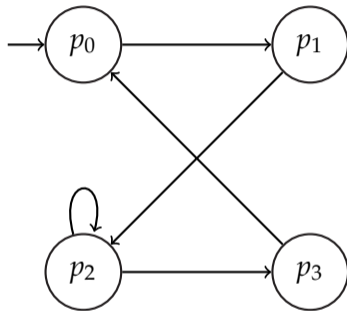
Examples:

- $G(\bigwedge_{i=0}^3 (p_i \rightarrow \bigvee_{\substack{j=0 \\ j \neq i}}^3 p_j))$
 - False
 - Counterexample: $\langle p_0, p_1, p_2, p_2 \rangle$
- $F(p_3)$



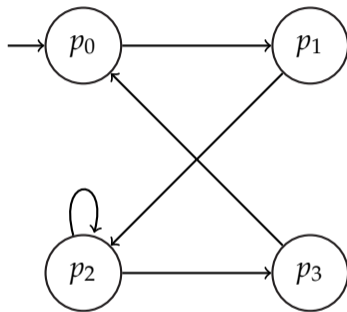
Examples:

- $$G\left(\bigwedge_{i=0}^3 (p_i \rightarrow \bigvee_{\substack{j=0 \\ j \neq i}}^3 p_j)\right)$$
 - False
 - Counterexample: $\langle p_0, p_1, p_2, p_2 \rangle$
- $$F(p_3)$$
 - False
 - Counterexample: $\langle p_0, p_1 \rangle \cdot (p_2)^\omega$



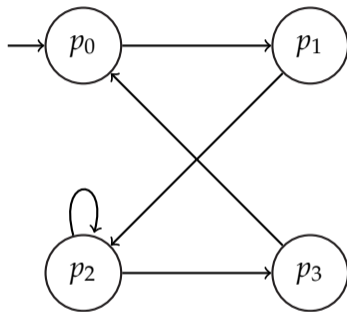
Examples:

- $G(\bigwedge_{i=0}^3 (p_i \rightarrow \bigvee_{\substack{j=0 \\ j \neq i}}^3 p_j))$
 - False
 - Counterexample: $\langle p_0, p_1, p_2, p_2 \rangle$
- $F(p_3)$
 - False
 - Counterexample: $\langle p_0, p_1 \rangle \cdot (p_2)^\omega$
- $FG(p_2)$



Examples:

- $G(\bigwedge_{i=0}^3 (p_i \rightarrow X \bigvee_{\substack{j=0 \\ j \neq i}}^3 p_j))$
 - False
 - Counterexample: $\langle p_0, p_1, p_2, p_2 \rangle$
- $F(p_3)$
 - False
 - Counterexample: $\langle p_0, p_1 \rangle \cdot (p_2)^\omega$
- $FG(p_2)$
 - False
 - Counterexample: $(\langle p_0, p_1, p_2, p_3 \rangle)^\omega$





A multi-process example

```
1 process Inc =  
2   while true do  
3     if x < 200 then  
4       x := x + 1  
5   od  
6
```

```
1 process Dec =  
2   while true do  
3     if x > 0 then  
4       x := x-1  
5   od  
6
```

```
1 process Reset =  
2   while true do  
3     if x = 200 then  
4       x := 0  
5   od  
6
```



A multi-process example

```
1 process Inc =  
2   while true do  
3     if x < 200 then  
4       x := x + 1  
5   od  
6
```

```
1 process Dec =  
2   while true do  
3     if x > 0 then  
4       x := x - 1  
5   od  
6
```

```
1 process Reset =  
2   while true do  
3     if x = 200 then  
4       x := 0  
5   od  
6
```

Property: *"is x always between (and including) 0 and 200?"*



A multi-process example

```
1 process Inc =  
2   while true do  
3     if x < 200 then  
4       x := x + 1  
5   od
```

```
1 process Dec =  
2   while true do  
3     if x > 0 then  
4       x := x-1  
5   od
```

```
1 process Reset =  
2   while true do  
3     if x = 200 then  
4       x := 0  
5   od
```

- Suppose $x = 199$.



A multi-process example

```
1 process Inc =  
2   while true do  
3     if x < 200 then  
4       x := x + 1  
5   od
```

```
1 process Dec =  
2   while true do  
3     if x > 0 then  
4       x := x-1  
5   od
```

```
1 process Reset =  
2   while true do  
3     if x = 200 then  
4       x := 0  
5   od
```

- Suppose $x = 199$.



A multi-process example

```
1 process Inc =  
2   while true do  
3     if x < 200 then  
4       x := x + 1  
5     od  
6
```

```
1 process Dec =  
2   while true do  
3     if x > 0 then  
4       x := x-1  
5     od  
6
```

```
1 process Reset =  
2   while true do  
3     if x = 200 then  
4       x := 0  
5     od  
6
```

- Suppose $x = 199$.



A multi-process example

```
1 process Inc =  
2   while true do  
3     if x < 200 then  
4       x := x + 1  
5     od
```

```
1 process Dec =  
2   while true do  
3     if x > 0 then  
4       x := x-1  
5     od
```

```
1 process Reset =  
2   while true do  
3     if x = 200 then  
4       x := 0  
5     od
```

- Suppose $x = 199$.



A multi-process example

```
1 process Inc =  
2   while true do  
3     if x < 200 then  
4       x := x + 1  
5     od
```

```
1 process Dec =  
2   while true do  
3     if x > 0 then  
4       x := x-1  
5     od
```

```
1 process Reset =  
2   while true do  
3     if x = 200 then  
4       x := 0  
5     od
```

- Suppose $x = 199$.



A multi-process example

```
1 process Inc =  
2   while true do  
3     if x < 200 then  
4       x := x + 1  
5     od
```

```
1 process Dec =  
2   while true do  
3     if x > 0 then  
4       x := x-1  
5     od
```

```
1 process Reset =  
2   while true do  
3     if x = 200 then  
4       x := 0  
5     od
```

- Suppose $x = 199$.



A multi-process example

```
1 process Inc =  
2   while true do  
3     if x < 200 then  
4       x := x + 1  
5     od
```

```
1 process Dec =  
2   while true do  
3     if x > 0 then  
4       x := x-1  
5     od
```

```
1 process Reset =  
2   while true do  
3     if x = 200 then  
4       x := 0  
5     od
```

- Suppose $x = 199$.



A multi-process example

```
1 process Inc =  
2   while true do  
3     if x < 200 then  
4       x := x + 1  
5     od  
6
```

```
1 process Dec =  
2   while true do  
3     if x > 0 then  
4       x := x-1  
5     od  
6
```

```
1 process Reset =  
2   while true do  
3     if x = 200 then  
4       x := 0  
5     od  
6
```

- Suppose $x = 199$.
- Counterexample to the property: after 6 steps, x becomes strictly less than 0.
- How can we fix the error?



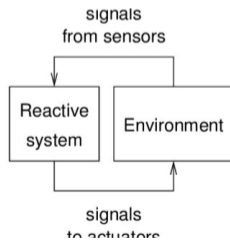
- ① Modeling phase
 - model the system under consideration
 - as a first sanity check, perform some simulations
 - formalise the property to be checked and perform sanity check on it
- ② Running phase
 - run the model checker to check the validity of the property in the model
- ③ Analysis phase
 - property satisfied? \Rightarrow check next property (if any)
 - property violated? \Rightarrow
 - analyse generated counterexample by simulation
 - refine the model, design, or property and repeat the entire procedure
 - out of memory? \Rightarrow try to reduce the model and try again



Pros of Model Checking

- widely applicable (hardware, software, protocol systems, ...)
- allows for partial verification (only most relevant properties)
- potential “push-button” technology (software-tools)
- rapidly increasing industrial interest
- in case of property violation, a counterexample is provided
- sound and interesting mathematical foundations
- not biased to the most possible scenarios (such as testing)

- main focus on control-intensive applications (less data-oriented)
 - reactive systems
- model checking is only as “good” as the system model
- no guarantee about completeness of results
- impossible to check generalisations (in general)





Notable model checking applications

- Security: Needham-Schroeder encryption protocol
 - error that remained undiscovered for 17 years unrevealed
- Transportation systems
 - train model containing 10^{476} states
- Model checkers for C, Java and C++
 - used (and developed) by Microsoft, Digital, NASA, FBK
 - successful application area: device drivers
- Software in the current/next generation of space missiles
 - NASA's Mars Pathfinder, Deep Space-1, JPL LARS group



Classical approach

In order to decide if $M, s \models \phi$:

- 1 Build the Büchi automaton \mathcal{A}_M that accepts all and only the words corresponding to computations of M starting from s ;
- 2 Build the Büchi automaton $\mathcal{A}_{\neg\phi}$ that accepts all and only the words corresponding to models of $\neg\phi$;
- 3 Check the *(non)emptiness* of the product automaton $\mathcal{A}_M \times \mathcal{A}_{\neg\phi}$.
 - $L(\mathcal{A}_M \times \mathcal{A}_{\neg\phi}) \neq \emptyset \Leftrightarrow M, s \not\models \phi$
 - MC=universal problem
 - EMPTINESS= existential problem



Classical approach

- Emptiness of $\mathcal{A}_M \times \mathcal{A}_{\neg\phi}$:
 - \exists final state $q . q_0 \rightsquigarrow q \wedge q \rightsquigarrow q$
- Checking the existence of a *fair cycle* in M
- **IMPORTANT**: in practice, this is much more difficult than simply the reachability of a state q .



Invariance checking

- **Invariance checking**: it is defined as LTL model checking of a formula of the form $G(\phi)$ where ϕ is a Boolean formula.

Does ϕ hold in (at least) every reachable state of M ?

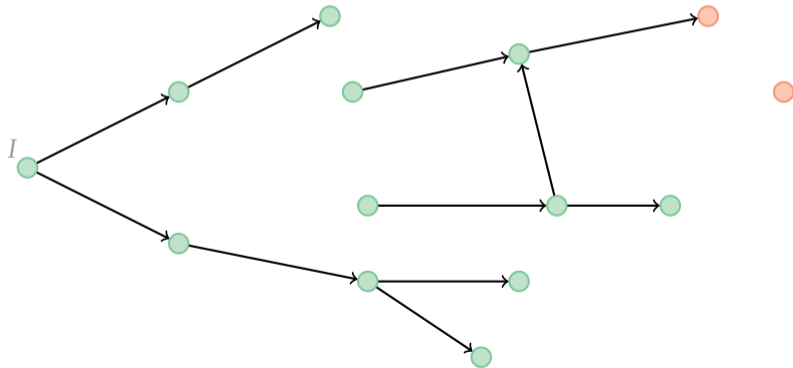
- Given $M = \langle \mathcal{AP}, Q, I, T, L \rangle$ and a Boolean formula ϕ over the variables \mathcal{AP}
find a reachable state in which $\neg\phi$ holds or establish its nonexistence.
 - *it is a reachability problem*
 - if ϕ holds in every reachable state of M , then ϕ is **invariant** in M
 - otherwise, there is a *finite trace* as counterexample:

$$\langle s_0, s_1, \dots, s_n \rangle$$

such that $s_i \models \phi$ for any $i < n$ and $s_n \not\models \phi$.

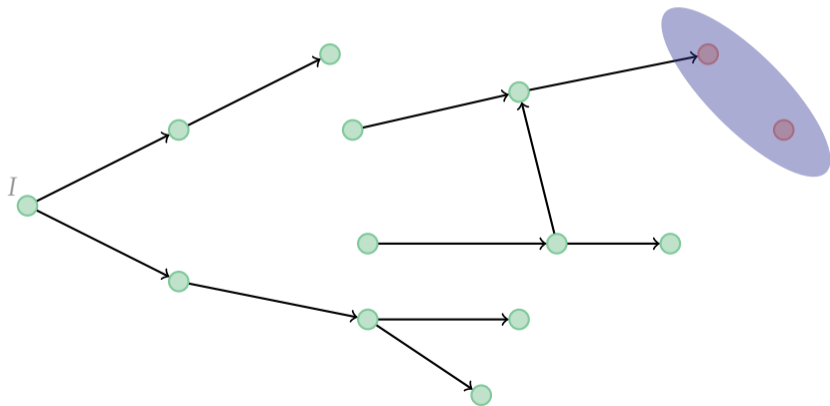


Standard Algorithm for Invariance Checking



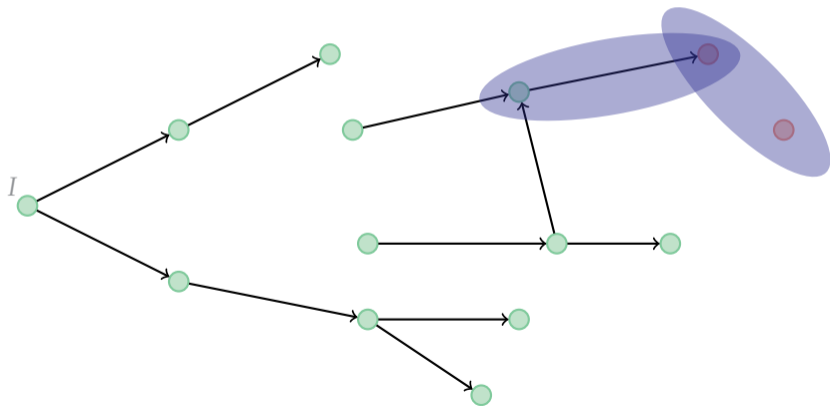


Standard Algorithm for Invariance Checking

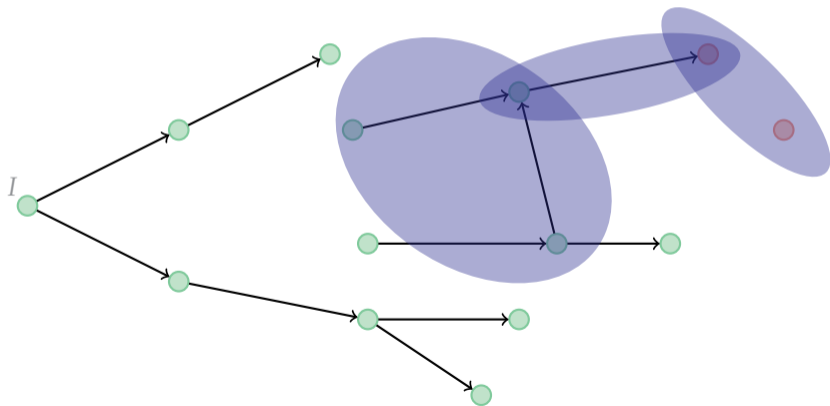




Standard Algorithm for Invariance Checking

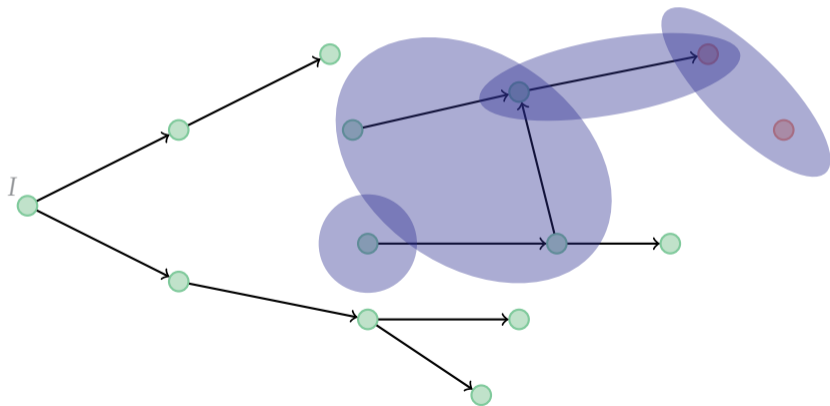


Standard Algorithm for Invariance Checking

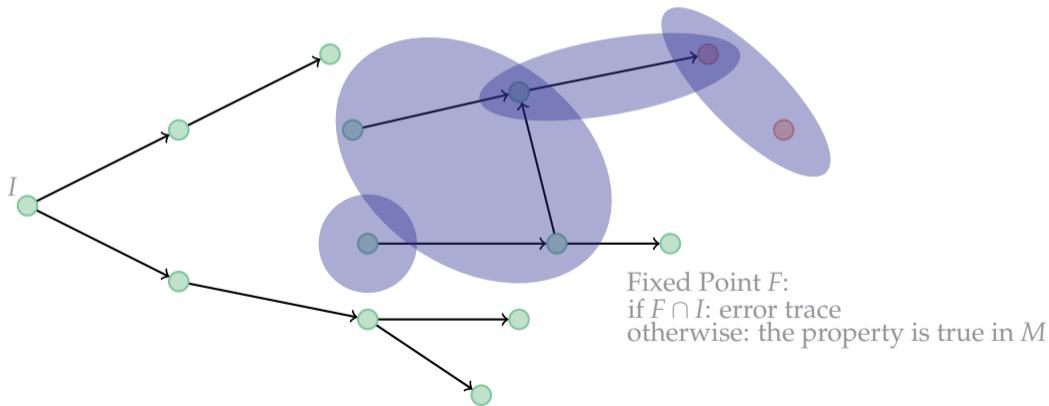




Standard Algorithm for Invariance Checking

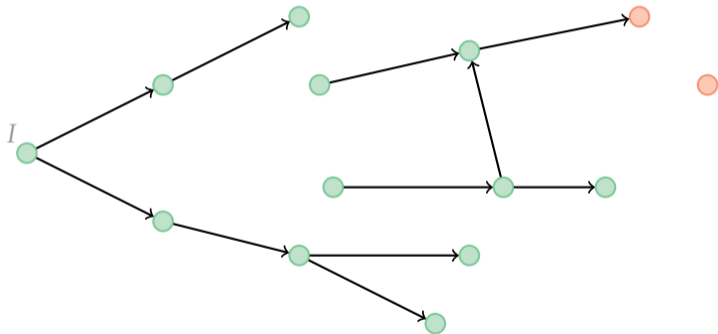


Standard Algorithm for Invariance Checking



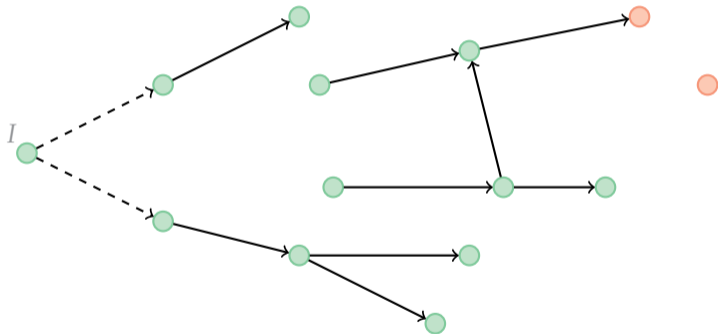


Standard Algorithm for Invariance Checking





Standard Algorithm for Invariance Checking



REFERENCES



- Alessandro Artale et al. (2023).** “Complexity of Safety and coSafety Fragments of Linear Temporal Logic”. In: *Proc. of the 36th AAAI Conf. on Artificial Intelligence*. AAAI Press.
- Giuseppe De Giacomo and Moshe Y. Vardi (2013).** “Linear Temporal Logic and Linear Dynamic Logic on Finite Traces”. In: *Proceedings of the 23rd International Joint Conference on Artificial Intelligence*. Ed. by Francesca Rossi. IJCAI/AAAI, pp. 854–860.
- A Prasad Sistla and Edmund M Clarke (1985).** “The complexity of propositional linear temporal logics”. In: *Journal of the ACM (JACM)* 32.3, pp. 733–749. DOI: 10.1145/3828.3837.