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

An introduction to Model Checking

Luca Geatti
luca.geatti@uniud.it
Angelo Montanari
angelo.montanari@uniud.it

April 30th, 2024

MODEL CHECKING

an introduction

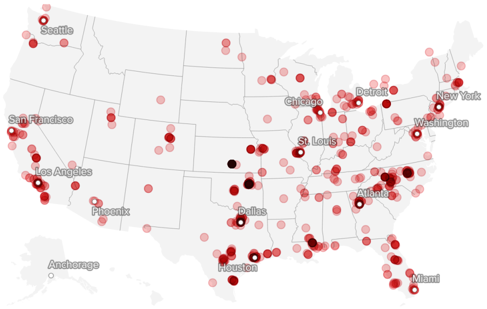
based on the slides of Prof. Joost-Pieter Katoen

- 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.

Model Checking
An introduction

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.

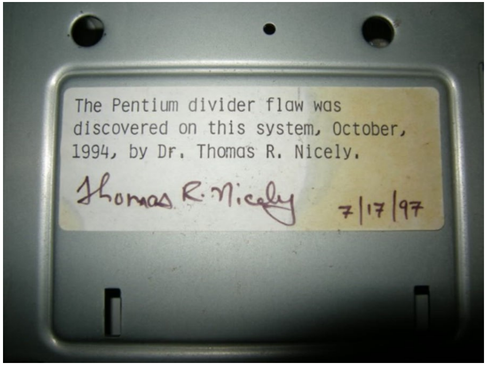


Note: Data as of 10 a.m. EST on Feb. 22, 2024.

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)



The Pentium divider flaw was discovered on this system, October, 1994, by Dr. Thomas R. Nicely.

Thomas R. Nicely 7/17/97

- 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.

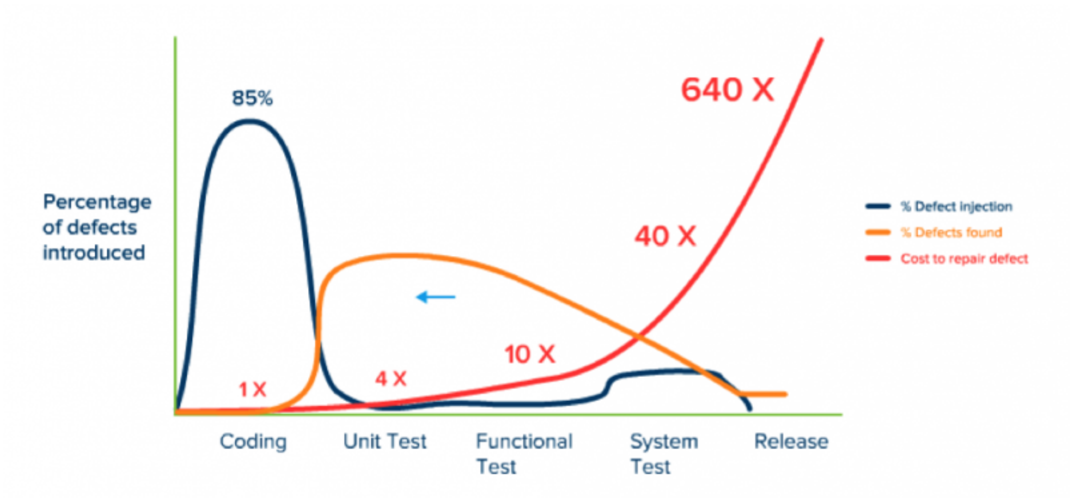
Unit Testing:

- 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.*

Formal Methods

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

Model-based simulation or testing

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

Deductive Methods
(based on proof rules)

$$\{p\}\ \mathbf{skip}\ \{p\}$$

$$\{p[x \mapsto t]\}\ x := t\ \{p\} \tag{2.54}$$

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

$$\frac{\{p \wedge b\}\ S_1\ \{q\} \qquad \{p \wedge \neg b\}\ S_2\ \{q\}}{\{p\}\ \mathbf{if}\ b\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \{q\}}$$

$$\frac{\{p \wedge b\}\ S\ \{p\}}{\{p\}\ \mathbf{while}\ b\ \mathbf{do}\ S\ \{p \wedge \neg b\}} \tag{2.55}$$

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

Figure 2.1: Hoare Logic for Partial Correctness – Syntactic Presentation

Milestones in Formal Methods

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.

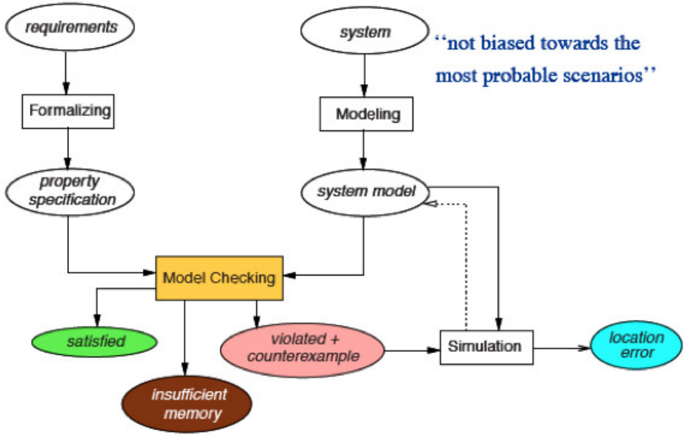
1980 (Clarke, Emerson, ...) Automated verification of concurrent programs. Model-based instead of proof-rule based approach. Does the concurrent program satisfy a given (logical) property?

Model Checking

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.

Model Checking



"not biased towards the most probable scenarios"

Model Checking
Turing Award 2007

(a) E.M. Clarke

(b) E.A. Emer-son

(c) J. Sifakis

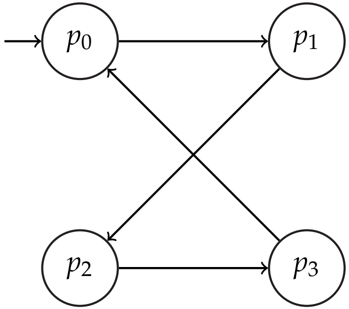
For their role in developing Model-Checking into a highly effective verification technology that is widely adopted in the hardware and software industries.

Model Checking
What is a model?

Definition (Kripke structure)

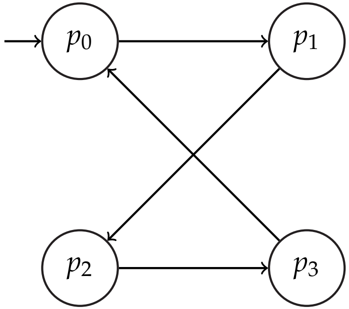
A Kripke structure (or Transition System) is a tuple $M = \langle \mathcal{AP}, Q, I, T, L \rangle$ where:

- $\mathcal{AP}$ 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
- $L : Q \to 2^{\mathcal{AP}}$ is the labeling function that assigns to each state the set of atoms in $\mathcal{AP}$ that are true in that state.



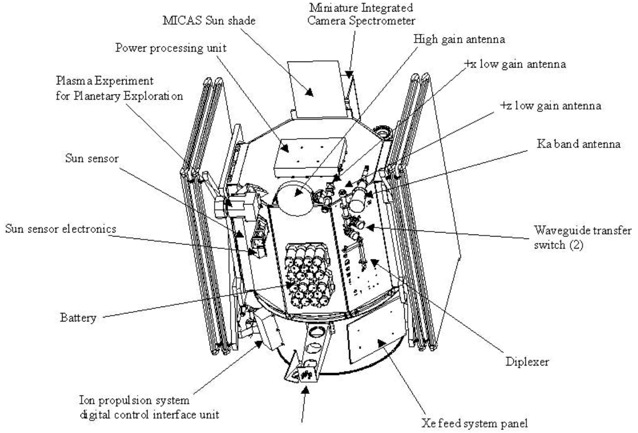
Expressivity:

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


$$\begin{tikzcd}
\arrow[r] & p_0 \arrow[r] & p_1 \arrow[dl] \\
& p_2 \arrow[r] & p_3 \arrow[ul]
\end{tikzcd}$$

Model Checking
What is a model?

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)

Model Checking

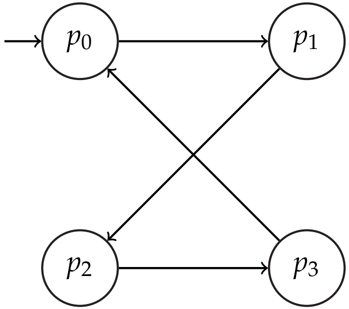
Definition (Model Checking of LTL)

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 $M, s \models \mathsf{A}\phi$ iff *all* paths of M starting from s are models of ϕ.

A is the *"for all paths"* operator of CTL.


$$\begin{tikzcd}
\arrow[r] & p_0 \arrow[r] & p_1 \arrow[dl] \\
& p_2 \arrow[r] & p_3 \arrow[ul]
\end{tikzcd}$$

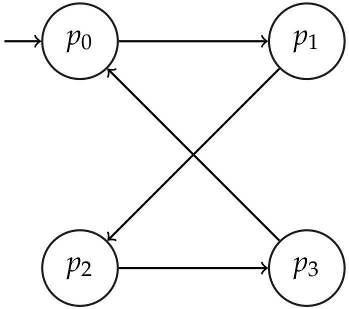
Model Checking

Definition (Model Checking of LTL)

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

Example:

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


$$\begin{tikzcd} \arrow[r] & p_0 \arrow[r] & p_1 \arrow[dl] \\ & p_2 \arrow[r] & p_3 \arrow[ul] \end{tikzcd}$$

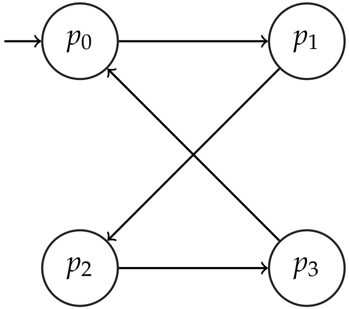
Model Checking

Theorem

The LTL-*MC is* PSPACE-*complete.*

Reference:

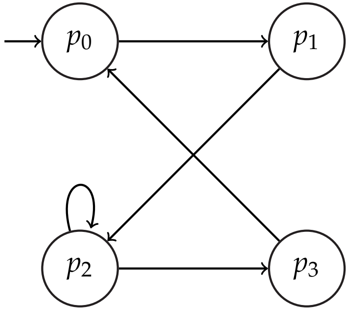
A Prasad Sistla and Edmund M Clarke (1985). "The complexity of propositional linear temporal logics". In: *Journal of the ACM (JACM)* 32.3, pp. 733–749. DOI: 10.1145/3828.3837


$$\begin{tikzcd}
\arrow[r] & p_0 \arrow[r] & p_1 \arrow[dl] \\
& p_2 \arrow[r] & p_3 \arrow[ul]
\end{tikzcd}$$

Model Checking
More examples

Examples:

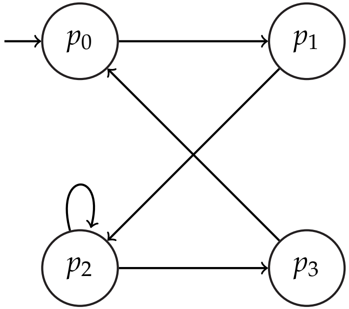
- $\mathsf{G}(\bigwedge_{i=0}^{3}(p_i \rightarrow \mathsf{X} \bigvee_{\substack{j=0\\j\neq i}}^{3} p_j))$



Model Checking
More examples

Examples:

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

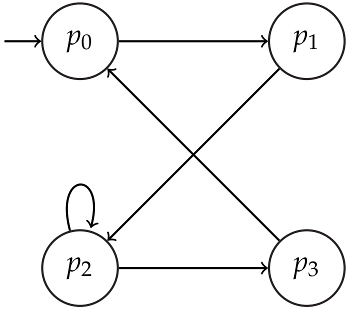


Model Checking

More examples

Examples:

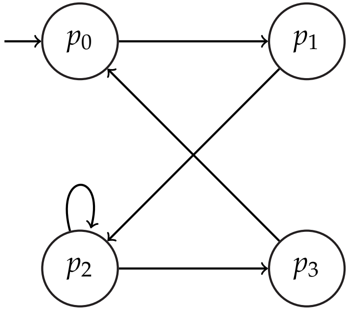
- $\mathsf{G}(\bigwedge_{i=0}^{3}(p_i \rightarrow \mathsf{X} \bigvee_{\substack{j=0 \\ j\neq i}}^{3} p_j))$
  - False
  - Counterexample: $\langle p_0, p_1, p_2, p_2 \rangle$
- $\mathsf{F}(p_3)$


$$\begin{tikzcd}
\arrow[r] & p_0 \arrow[r] & p_1 \arrow[dl] \\
& p_2 \arrow[loop above] \arrow[r] & p_3 \arrow[ul]
\end{tikzcd}$$

Model Checking
More examples

Examples:

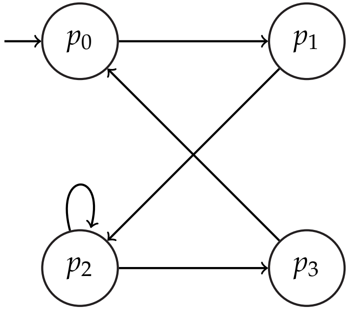
- $\mathsf{G}(\bigwedge_{i=0}^{3}(p_i \to \mathsf{X} \bigvee_{\substack{j=0 \\ j\neq i}}^{3} p_j))$
  - False
  - Counterexample: $\langle p_0, p_1, p_2, p_2\rangle$
- $\mathsf{F}(p_3)$
  - False
  - Counterexample: $\langle p_0, p_1\rangle \cdot (p_2)^\omega$



Model Checking
More examples

Examples:

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

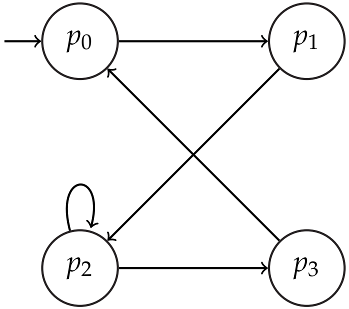


$$\begin{tikzcd}
\arrow[r] & p_0 \arrow[rr] & & p_1 \arrow[dll] \\
& p_2 \arrow[rr] \arrow[loop above] & & p_3 \arrow[ull]
\end{tikzcd}$$

Model Checking
More examples

Examples:

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



A multi-process example

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

```

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

```

```
process Reset =
  while true do
    if x = 200 then
      x := 0
  od

```

A multi-process example

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

```

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

```

```
process Reset =
  while true do
    if x = 200 then
      x := 0
  od

```

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

A multi-process example

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

```

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

```

```
process Reset =
  while true do
    if x = 200 then
      x := 0
  od

```

- Suppose $x = 199$.

A multi-process example

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

```

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

```

```
process Reset =
  while true do
    if x = 200 then
      x := 0
  od

```

- Suppose $x = 199$.

A multi-process example

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

```

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

```

```
process Reset =
  while true do
    if x = 200 then
      x := 0
  od

```

- Suppose $x = 199$.

A multi-process example

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

```

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

```

```
process Reset =
  while true do
    if x = 200 then
      x := 0
  od

```

- Suppose $x = 199$.

A multi-process example

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

```

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

```

```
process Reset =
   while true do
      if x = 200 then
         x := 0
   od

```

- Suppose $x = 199$.

A multi-process example

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

```

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

```

```
process Reset =
   while true do
      if x = 200 then
         x := 0
   od

```

- Suppose $x = 199$.

A multi-process example

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

```

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

```

```
process Reset =
  while true do
    if x = 200 then
      x := 0
  od

```

- Suppose $x = 199$.

A multi-process example

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

```

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

```

```
process Reset =
  while true do
    if x = 200 then
      x := 0
  od

```

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

The Model Checking Process

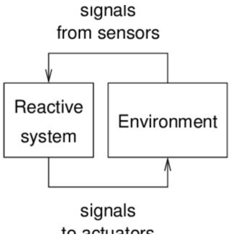
1. Modeling phase
   - model the system under consideration
   - as a first sanity check, perform some simulations
   - formalise the property to be checked and perform sanity check on it
2. Running phase
   - run the model checker to check the validity of the property in the model
3. Analysis phase
   - 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)

Cons of Model Checking

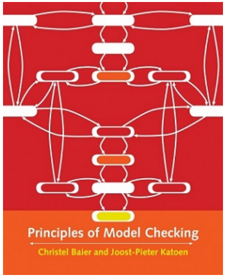
- main focus on control-intensive applications (less data-oriented)
  - reactive systems
- model checking is only as "good" as the system model
- no guarantee about completeness of results
- impossible to check generalisations (in general)

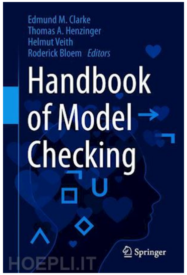


Notable model checking applications

- Security: Needham-Schroeder encryption protocol
  - error that remained undiscovered for 17 years unrevealed
- Transportation systems
  - train model containing 10^{476} states
  - state-space explosion problem
- 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

Two excellent and modern books
on model checking


Principles of Model Checking
Christel Baier and Joost-Pieter Katoen



Reference

Christel Baier and Joost-Pieter Katoen (2008). ***Principles of model checking.*** **MIT Press. ISBN: 978-0-262-02649-9**

Reference

Edmund M Clarke et al. (2018). ***Handbook of model checking.*** **Vol. 10. Springer. DOI: 10.1007/978-3-319-10575-8**

REFERENCES

Bibliography I

Christel Baier and Joost-Pieter Katoen (2008). ***Principles of model checking*.** MIT Press. ISBN: 978-0-262-02649-9.

Edmund M Clarke et al. (2018). ***Handbook of model checking*.** Vol. 10. Springer. DOI: 10.1007/978-3-319-10575-8.

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.