Rigorous Methods for Software Engineering (F21RS-F20RS) Automata Based Model Checking: How It Works (Part 2)

Andrew Ireland
Department of Computer Science
School of Mathematical and Computer Sciences
Heriot-Watt University
Edinburgh

Overview

- The verification problem.
- The model checking solution to the verification problem.

A Verification Problem

$$M, S_0 \models P$$

- Informally, $\boldsymbol{M}$ denotes a **Promela model** with initial state $\boldsymbol{S_0}$ while $\boldsymbol{P}$ denotes an **LTL property**.
- The logical operator $\models$ means **"... satisfies ..."**, it is analogous to saying *"program M executes correctly for test case P."*
- But model checking is much more powerful than testing, i.e. **it is equivalent to exhaustive testing!**
- If the above statement is correct, then starting in state $\boldsymbol{S_0}$, **ALL** possible executions of $\boldsymbol{M}$ satisfy $\boldsymbol{P}$.

A Verification Problem – An Algorithmic Solution

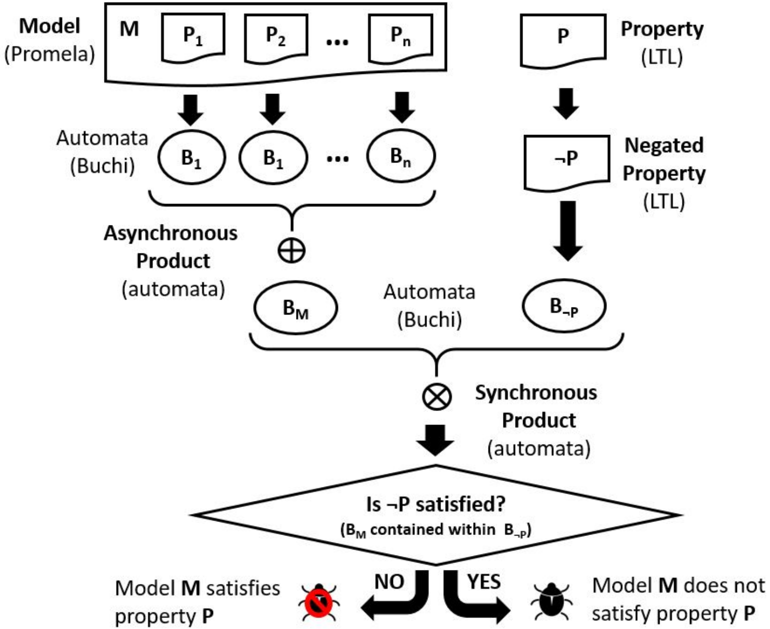
$$M, S_0 \models P$$

- Prove $\boldsymbol{P}$ by searching for a **counter-example**, i.e. a path from the initial state $\boldsymbol{S_0}$ to a state within $\boldsymbol{M}$ where $\neg\boldsymbol{P}$ is true.
- If a **counter-example** is **not** found then $\boldsymbol{P}$ is true

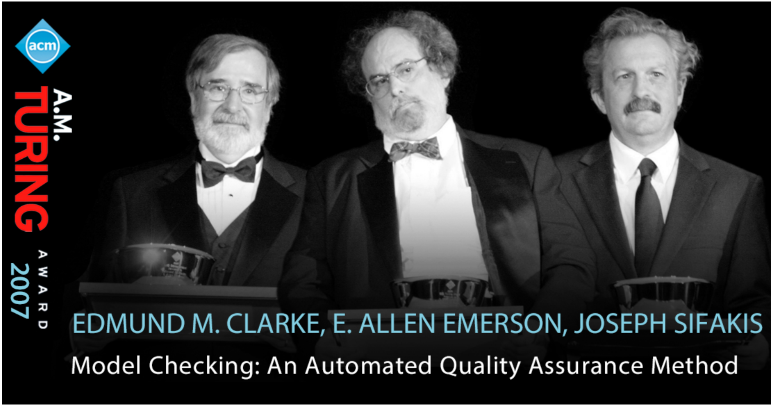
  else $\boldsymbol{P}$ is false.

An counter-example is a very useful aid for debugging, i.e. it denotes a simulation trace that illustrates a bug.

Model Checking – An Algorithmic Solution



Model Checking - An Algorithmic Solution



ACM 2007 Turing Award
Edmund Clarke, Allen Emerson, and Joseph Sifakis
Model Checking: Algorithmic Verification and Debugging

A Worked Example – Promela View

```
ltl R { [](x < 2) }

int x = 0;

active proctype P(){
do
:: !(x % 2)  -> x = x+1;    /* inc x when EVEN */
od}

active proctype Q(){
do
:: (x % 2)  -> x = x-1;     /* dec x when ODD */
od}
```

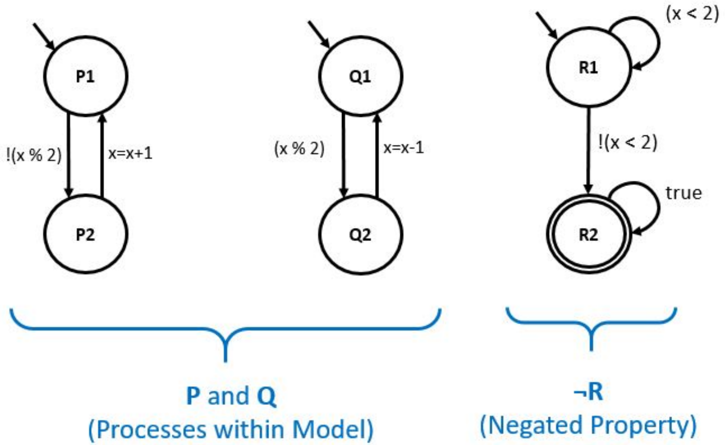
Some Useful Equivalence Properties

A negated property can be simplified using the following equivalences:

$$
\begin{aligned}
\neg\Box X &\leftrightarrow \Diamond\neg X \\
\neg\Diamond X &\leftrightarrow \Box\neg X \\
\neg(X \wedge Y) &\leftrightarrow \neg X \vee \neg Y \\
\neg(X \vee Y) &\leftrightarrow \neg X \wedge \neg Y \\
\neg(X \rightarrow Y) &\leftrightarrow X \wedge \neg Y
\end{aligned}
$$

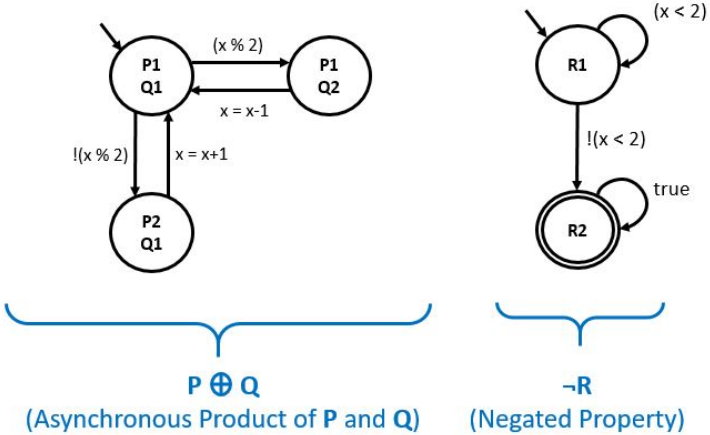
Note that $\neg X \equiv\ !X$ and $(X \rightarrow Y) \equiv\ (\neg X \vee Y)$

A Worked Example – Automata View

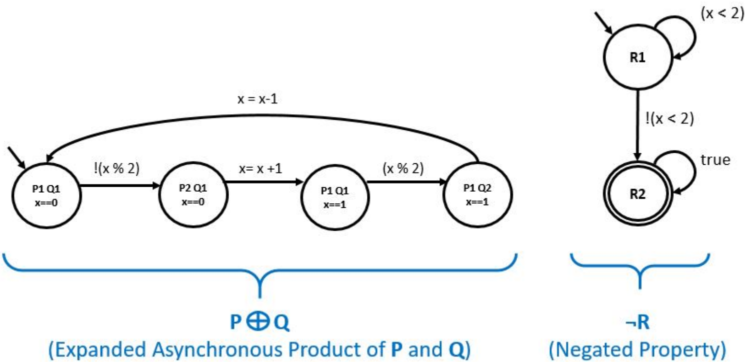


Note that $\neg R \equiv \neg\Box(x < 2) \equiv \Diamond\neg(x < 2) \equiv \Diamond!(x < 2)$

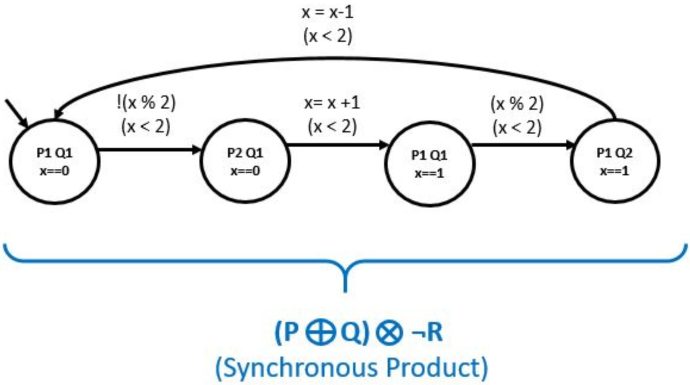
A Worked Example – Automata View



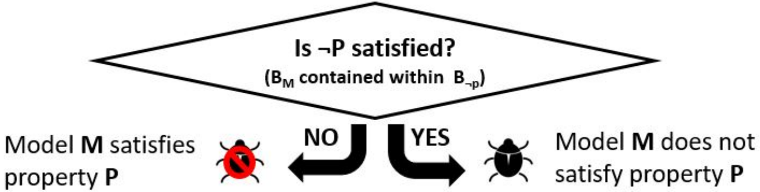
A Worked Example – Automata View



A Worked Example – Automata View



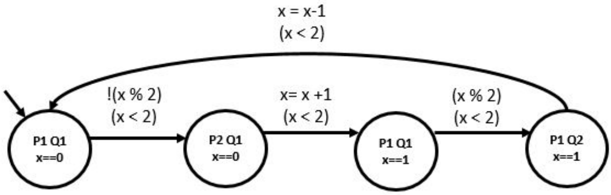
Verification Algorithm – Reminder



- **"contained within"** = there exists an infinite cycle through an **accept** state, a.k.a. an **acceptance cycle**.
- if **acceptance cycle** then **property** $\neg P$ is **satisfied**.

  else **property** P is **satisfied**.

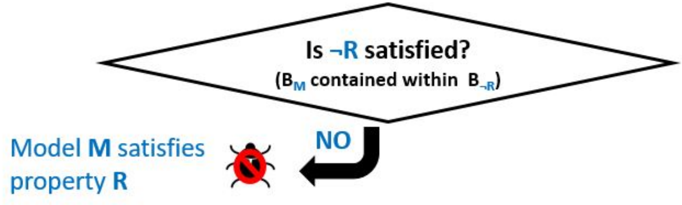
A Worked Example – Automata View



NO ACCEPTANCE CYCLES = Model **M** satisfies property **R**

Note: **R** is [](x < 2)

Verification Algorithm – Property **R** Satisfied



- **"contained within" is false** = no infinite cycle through an **accept** state, i.e. no **acceptance cycle**.
- if **acceptance cycle** then **property** $\neg R$ is **satisfied**.

  else **property** R is **satisfied**.

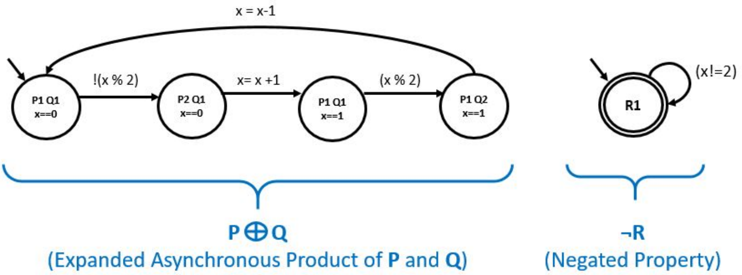
No acceptance cycle therefore **R** is **satisfied.**

The Worked Example – Revisited

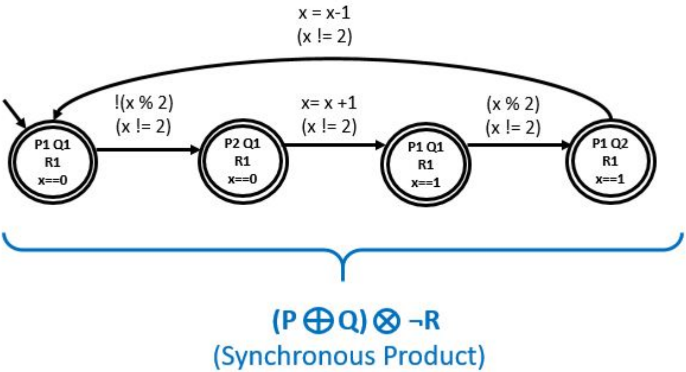
```
ltl R { <>(x == 2) }

int x = 0;

active proctype P(){
do
:: !(x % 2)  -> x = x+1;    /* inc x when EVEN */
od}

active proctype Q(){
do
:: (x % 2)  -> x = x-1;     /* dec x when ODD */
od}
```

- Same program but a new **R**, i.e. `<>(x == 2)`
- Note: $\neg \boldsymbol{R} \equiv \neg\Diamond(\boldsymbol{x} == 2) \;\equiv\; \Box\neg(\boldsymbol{x} == 2) \;\equiv\; \Box(\boldsymbol{x} != 2)$

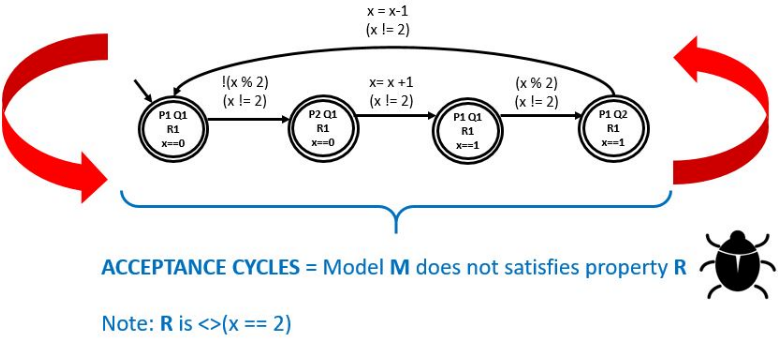
Automata Revisited



Synchronous Product – Revisited

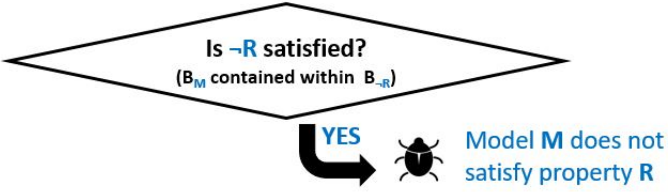


Automata Revisited



Note that the above Büchi automaton contains an **acceptance cycle**, i.e. a path that will infinitely often visit an **accept** state.

Verification Algorithm – Property **R** Not Satisfied



- **"contained within" is true** = an infinite cycle through an **accept** state, i.e. exists an **acceptance cycle**.
- if **acceptance cycle** then **property** $\neg R$ is **satisfied**.

  else **property** R is **satisfied**.

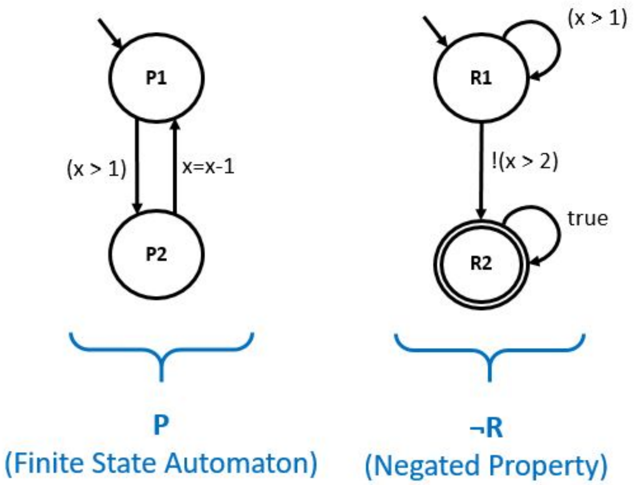
Acceptance cycle therefore R is **NOT satisfied**.

Stutter Steps Revisited

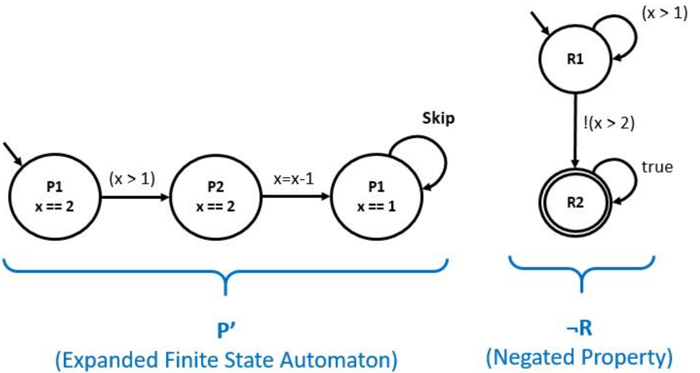
```
ltl R { <>(x > 1) }

byte x = 2;

active proctype P()
{
   do
   :: (x > 1) -> x = x-1;
   od
}
```

Note that process **P** deadlocks with **x** equal to **1**, i.e. process **P** represents a **finite computation**.

Stutter Steps Revisited

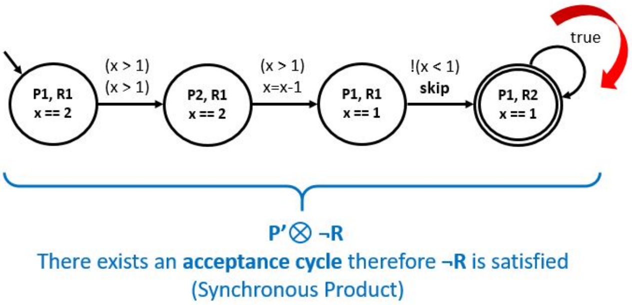


Stutter Steps Revisited



Note that the **skip** transition shown above (a.k.a. stutter step) turns deadlock, i.e. a finite computation, into an infinite computation with the same semantics.

Stutter Steps Revisited



Note that without the **skip** transition there would be **no acceptance cycle**. Note also that typically **stutter steps** are left implicit for presentation purposes.

The Limits of Model Checking

Problems:

- Model checking is limited to systems involving finite states, but in the real-world there are systems with infinite states, e.g. a simple integer counter.
- Even with finite state systems, the state space may become to large to represent – the so called **state explosion problem.**

Managing the problems:

- Building an **abstract** model will reduce the size of the state space, but in general **abstraction** is not automatic.
- Techniques that avoid having to have an explicit representation of all the execution paths, e.g.
  - **On-the-fly model checking:** incrementally explore execution paths.
  - **Symbolic model checking:** use of logical formulae to represent multiple states and transitions.
  - **Bounded model checking**, e.g. depth-first iterative deepening rather than depth first.

Summary

Learning outcomes:

- Understand the **model checking** algorithm and how to apply it to the verification of concurrent systems, i.e. systems involving multiple interacting processes.
- Understand the role that the **stutter step** plays in model checking.

Recommended reading:

- "Model Checking", E.M. Clarke, O. Grumberg, D.A. Peled, MIT Press, 1999.
- "Practical Formal Methods Using Temporal Logic", M. Fisher, Wiley, 2011.
- **Büchi Store:** `http://buchi.im.ntu.edu.tw/`