Formal Verification:
All Questions and Some Answers

A. Pnueli
Weizmann Institute of Sciences
The John von Neumann Minerva Center for the
Verification of Reactive Systems

CS Leading Teachers Course, WIS
9.5.99

The Programming Problem

Computers are entering more and more areas of our life.

Most Computer applications consist of an application-specific program (*software*) running on a standard hardware.

It is the program that drives the computer and executes the application.

A program is a list of instructions telling the computer what to do in very small steps.

Relatively small programs can generate an immense number of different behaviors.

The main obstacle to apply computers to an ever expanding spectrum of applications is our limited ability to construct absolutely reliable and error-free complex programs (and systems).

The Problem:

Find a way to construct absolutely reliable and error-free complex programs.

The Solution:

Use Mathematics!

The Programming Process: As it is done today

- Formulate requirements: What the system should and should not do.
- Specification and analysis.
- Design: How would the system achieve its goal.
- Coding (the actual programming).
- Unit testing.
- Integration and system testing.

Correctness is validated by *simulation* and *testing*, which

> Can only prove the presence of a bug, but never its absence!

[Dijsktra]

Drawbacks of Present Practice

- Validation by testing does not achieve the required reliability standard. In particular for safety-critical systems.

- A bug discovered by testing or in the field, may be extremely expensive to fix.

The Basic Notions

(Informal) Verification is the process of ensuring that the design is correct. Usually achieved by testing. Sometimes, refined into

- Validation – ensuring self-consistency and robustness.
- Verification – ensuring that the design conforms with the requirements.

Formal Verification – Ensuring that a formal model of the design satisfies a formal specification with mathematical certainty.

The long-term goal of the work on formal verification is to replace some of the testing by formal verification. Make sure that we explore all possible behaviors captured by the formal model.

Turning Programming into a Mathematical Problem

The general mathematical paradigm considers a *constraint* $C(x)$, e.g.

$$2 < x \leq 10$$

and asks questions such as:

- Does $x = 5$ satisfy the constraint?
- Is the constraint satisfiable by some x?
- Find an x which satisfies the constraint.
- Find the *best*, say *maximal*, x which satisfies C.

Question: If x is the program, what is C?

Answer: C is the specification which the program should satisfy.

Mathematical Specification of Programs

There are two classes of programs:

Computational Programs: Run in order to produce a final result on termination.

Can be modeled as a black box.

Specified in terms of Input/Output relations.

Example:

The program which computes

$$y = 1 + 3 + \cdots + (2x - 1)$$

Can be specified by the requirement

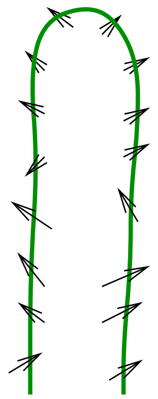
$$y = x^2.$$

Reactive Programs

Programs whose role is to maintain an ongoing interaction with their environments.

Examples: Air traffic control system, Programs controlling mechanical devices such as a train, a plane, or ongoing processes such as a nuclear reactor.

Can be viewed as a green cactus (?)



Such programs must be specified and verified in terms of their behaviors.

Milestone 1: Identification of reactive programs as a unique class that must be formalized in terms of the program's behaviors.

Temporal Logic

A mathematical language for specifying behaviors. Reference to time through modal operators rather than explicit parameterization.

Note that, in spoken language, we also say

Tomorrow I will take a trip

rather than

At $date = 10$ I take a trip

Main temporal operators are

$\square$	–	Always
$\diamondsuit$	–	Eventually

Example: Specification of an Elevator

- It is never the case that the doors are open while the elevator is in motion.

$$\square \neg(\mathit{doors\_open} \wedge \mathit{moving}).$$

- Any request for floor 5 is eventually satisfied.

$$\square (\mathit{request\_for}[5] \longrightarrow \diamond (\mathit{elevator\_at}[5] \wedge \mathit{doors\_open})).$$

Milestone 2: Develop Temporal Logic as a specification language for reactive systems.

The Temporal Framework for Reactive Systems Verification

- A computational model providing an abstract syntactic base for all reactive systems. There exists a hierarchy of such models.
- A Specification Language for specifying systems and their properties. Recommend temporal logic.
- An Implementation Language for describing proposed implementations (both software and hardware).
- Verification Techniques for validating that an implementation satisfies a specification. Practiced approaches:
  - Algorithmic verification methods for exploratory verification of finite-state systems: Enumerative and Symbolic variants.
  - A deductive methodology based on theorem-proving methods. Can accommodate infinite-state systems, but requires user interaction.

Temporal Verification

Prove that a temporally specified property holds over all computations of a system. Note, that there are infinitely many such computations.

How?

Algorithmic methods applicable to finite-state systems:

To prove that state property p holds over all computations,

- Construct all states reachable by the system.
- Check that each of them satisfies p.

Deductive methods use mathematical induction:

To prove that state property p holds over all computations, show that

- p holds over all initial states, and
- Every legal successor of each p-state is also a p-state.

A Simple Calculation

Assume a program with 10 parallel processes, each containing 5 statements.

The number of states reachable after termination is at least

$$25^{10} > 10^{13} = 10,000,000,000,000.$$

Testing or exhaustive examination requires that many steps.

Applying mathematical induction requires establishing 50 formulas (implications).

Milestone 3: Development of a verification methodology based on mathematical induction.

The Impossible Dream

We only tackle unsolvable problems.

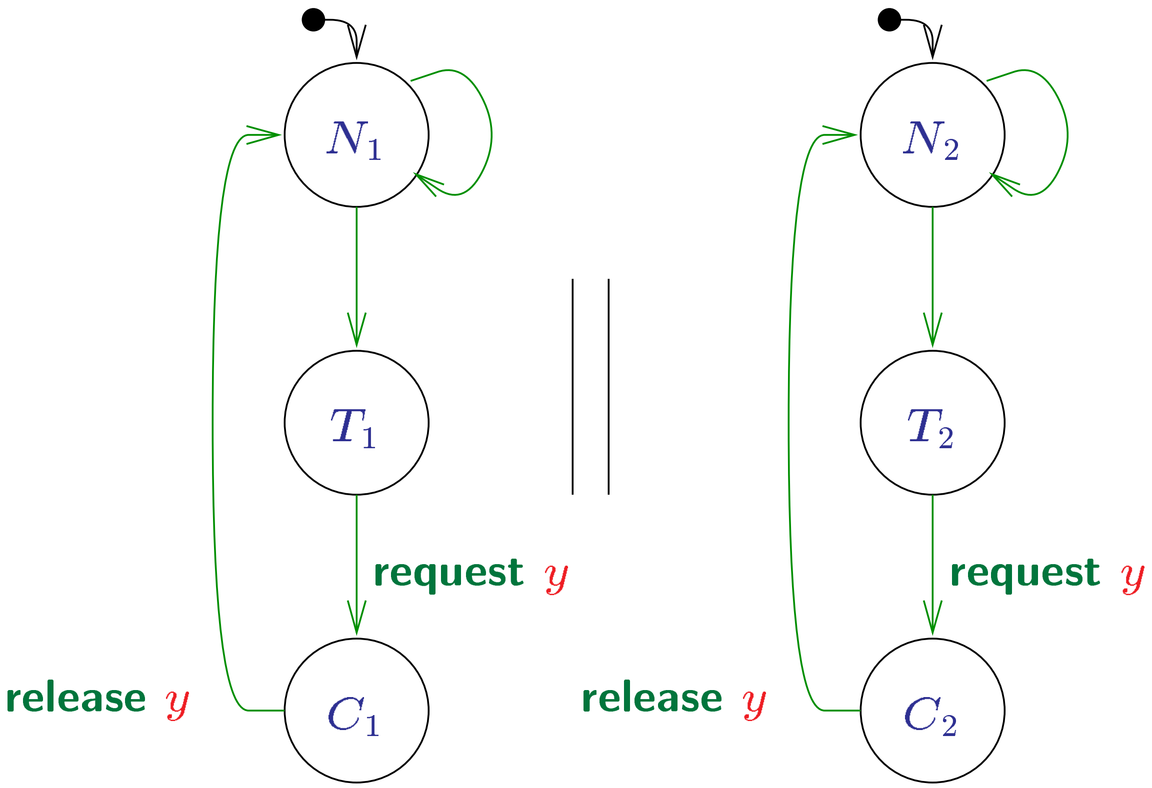
For the infinite-state case, the Verification Problem is undecidable.

For the finite-state case, the Verification Problem is P-SPACE complete — practically Exponential.

Example: Coordination by Semaphores

Two processes coordinating by Semaphores —

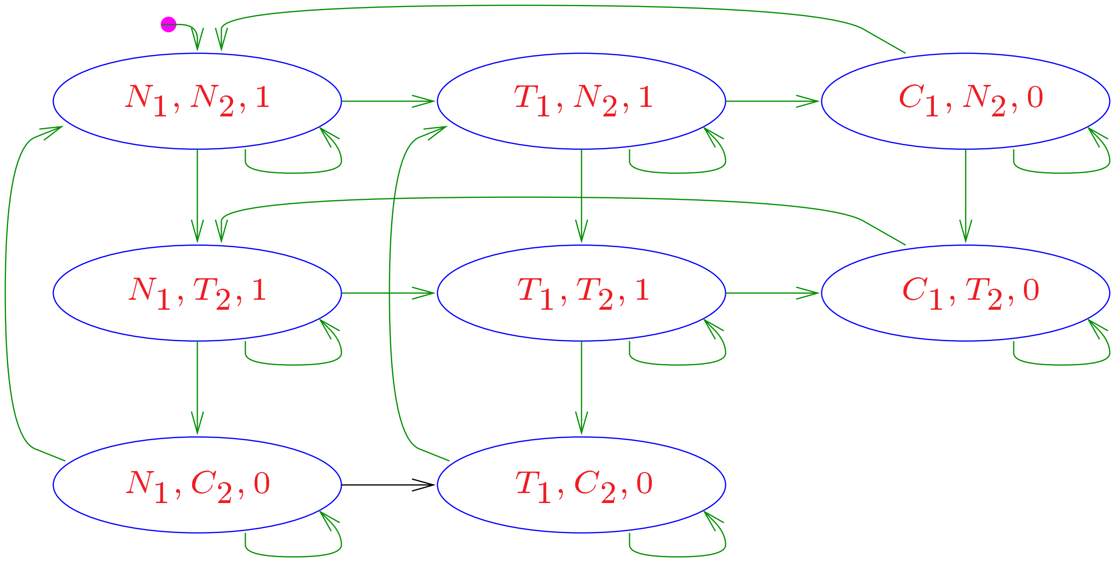
y: **integer initially** $y = 1$



The semaphore instructions **request** y and **release** y stand for

$$\langle \textbf{await } y > 0\,;\ y := y - 1 \rangle \quad \text{and} \quad y := y + 1.$$

The state-transition Graph for MUX-SEM



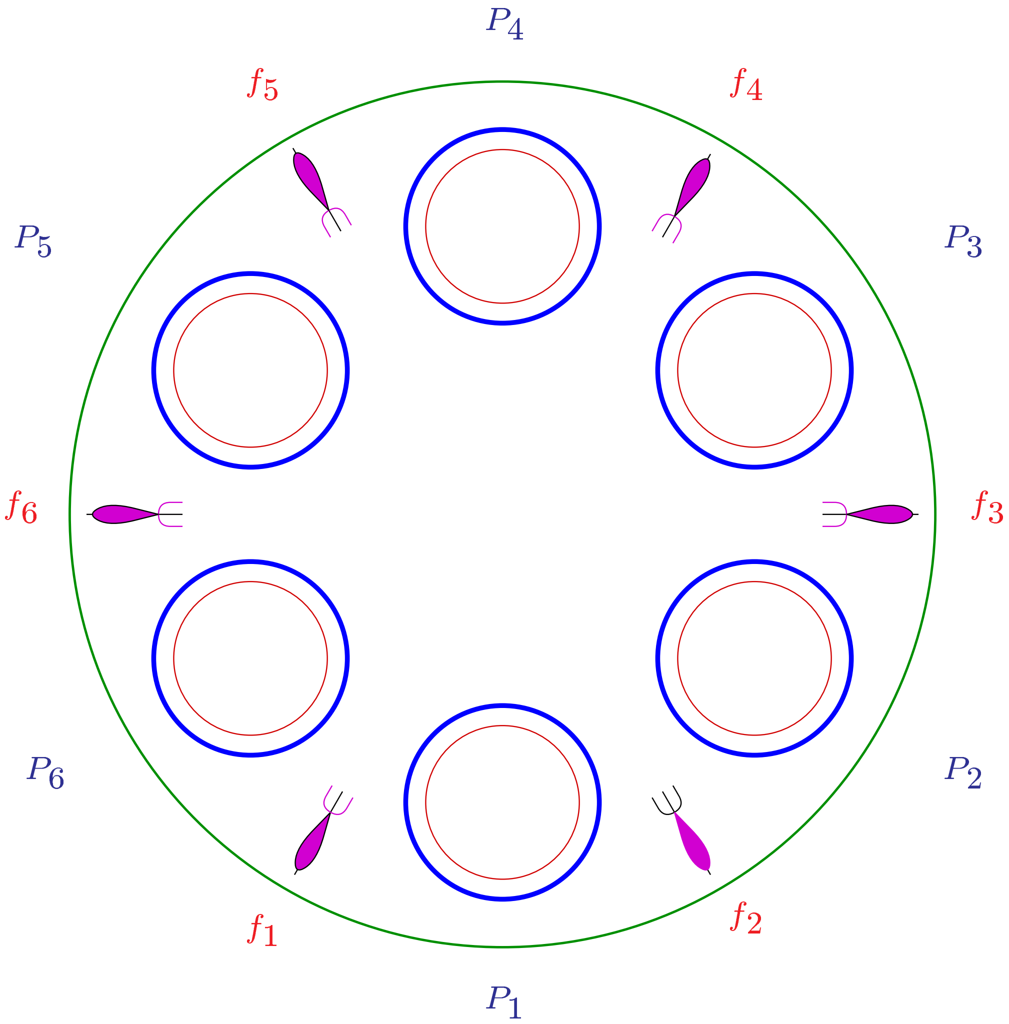
From this exhaustive enumeration, we conclude the validity of mutual exclusion:

$$\square\, \neg(C_1 \ \wedge \ C_2)$$

— No computation of the program can include a state in which process P_1 is at C_1 while P_2 is at C_2.

The Dining Philosophers Metaphor

Consider n philosophers arranged around a table.



The life of a philosopher alternates between a thinking phase (a non-critical activity) and an eating phase. In order to eat, a philosopher needs both forks.

Program Dine

A first attempt yields the following program Dine:

$$
\begin{array}{lll}
\textbf{in} & n & : \textbf{integer initially } n \geq 2 \\
\textbf{local} & f & : \textbf{array } [1..n] \textbf{ of integer initially } f = 1
\end{array}
$$

$$
\overset{n}{\underset{j=1}{\|}} P[j] :: \left[\begin{array}{l}
\ell_0 : \textbf{loop forever do} \\
\quad \left[\begin{array}{ll}
\ell_1 : & \textbf{Non-Critical} \\
\ell_2 : & \textbf{request } f[j] \\
\ell_3 : & \textbf{request } f[j \oplus_n 1] \\
\ell_4 : & \textbf{Critical} \\
\ell_5 : & \textbf{release } f[j] \\
\ell_6 : & \textbf{release } f[j \oplus_n 1]
\end{array} \right]
\end{array} \right]
$$

It is not difficult to verify the following safety property

$$\square \neg(at\_\ell_4[1] \ \wedge \ at\_\ell_4[2]),$$

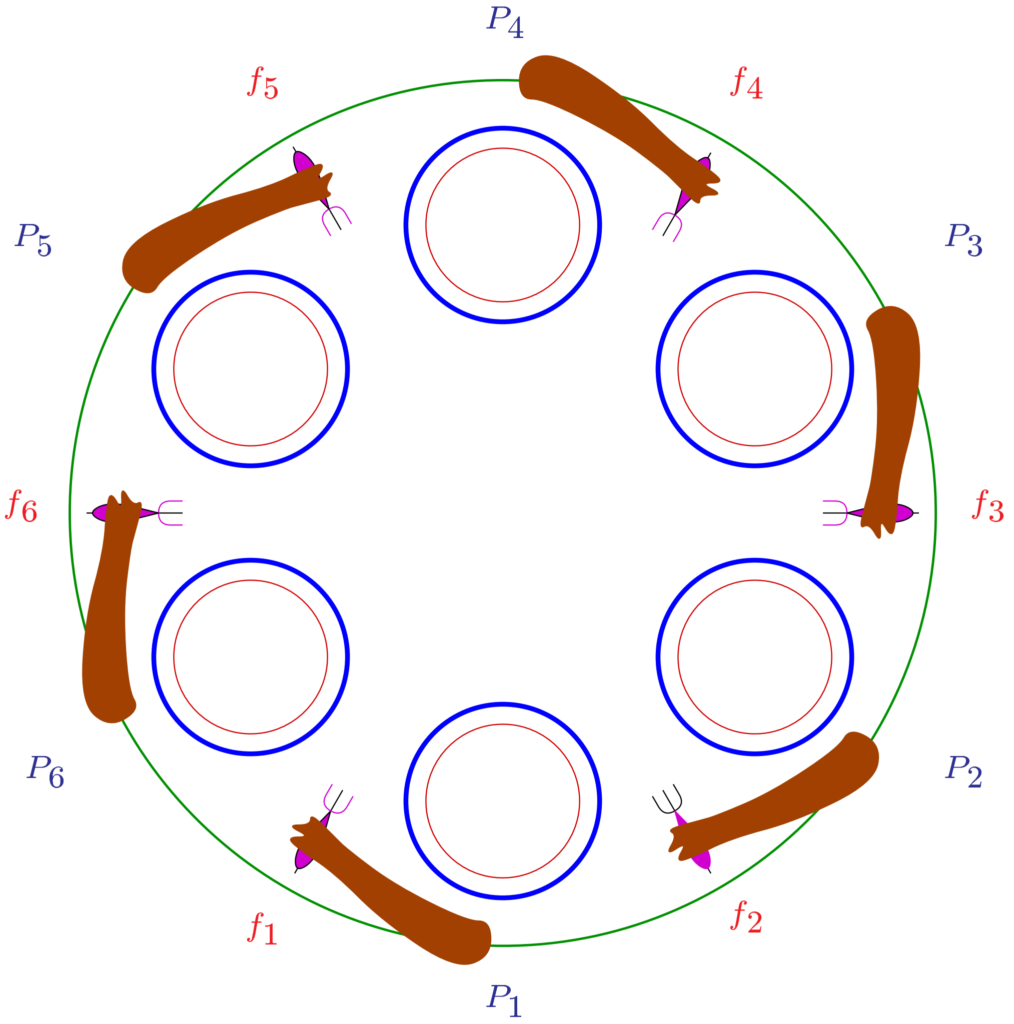
stating that philosophers $P[1]$ and $P[2]$ can never eat at the same time.

Accessibility not Guaranteed

Unfortunately, Dine cannot ensure accessibility for $P[1]$, specifiable by

$$\square\ (at\_\ell_2[1] \quad \rightarrow \quad \diamond\ at\_\ell_4[1])$$

Because all philosophers may deadlock together.



Solution: One Contrary Philosopher

$$
\begin{array}{lll}
\textbf{in} & n & : \textbf{integer initially } n \geq 2 \\
\textbf{local} & f & : \textbf{array } [1..n] \textbf{ of integer initially } f = 1
\end{array}
$$

$$
P[1] :: \left[\begin{array}{l}
\ell_0 : \textbf{loop forever do} \\
\quad \left[\begin{array}{ll}
\ell_1 : & \textbf{Non-Critical} \\
\ell_2 : & \textbf{request } f[2] \\
\ell_3 : & \textbf{request } f[1] \\
\ell_4 : & \textbf{Critical} \\
\ell_5 : & \textbf{release } f[2] \\
\ell_6 : & \textbf{release } f[1]
\end{array}\right]
\end{array}\right]
$$

$$\|$$

$$
\mathop{\Big\|}_{j=2}^{n} P[j] :: \left[\begin{array}{l}
\ell_0 : \textbf{loop forever do} \\
\quad \left[\begin{array}{ll}
\ell_1 : & \textbf{Non-Critical} \\
\ell_2 : & \textbf{request } f[j] \\
\ell_3 : & \textbf{request } f[j \oplus_n 1] \\
\ell_4 : & \textbf{Critical} \\
\ell_5 : & \textbf{release } f[j] \\
\ell_6 : & \textbf{release } f[j \oplus_n 1]
\end{array}\right]
\end{array}\right]
$$

Symbolic Finite-State Verification

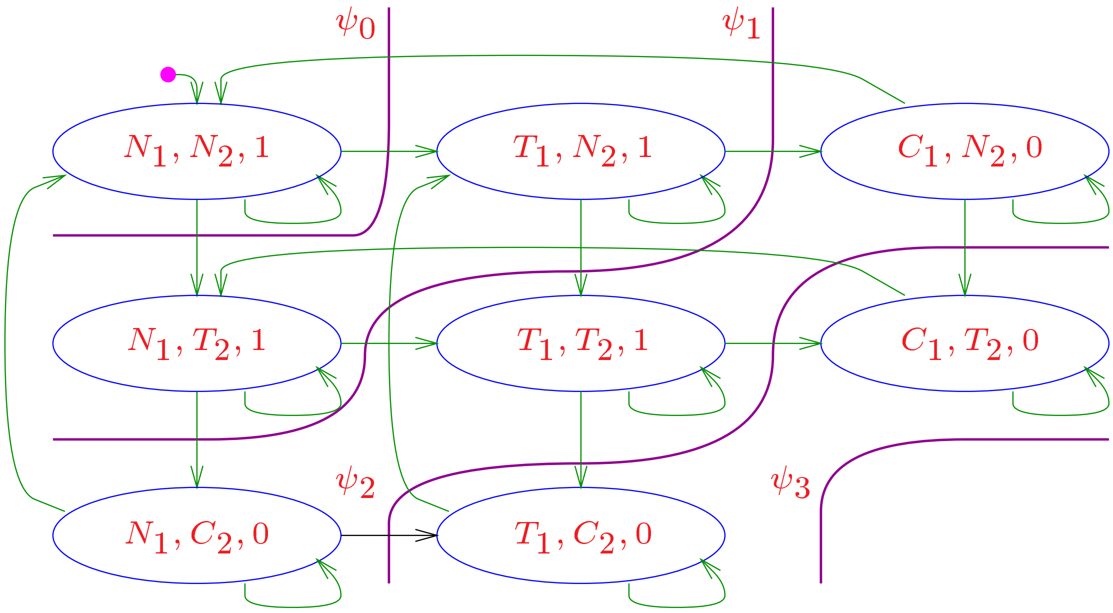
Enumerative methods can handle systems of sizes up to 10^7 ($\sim 2^{24}$) states. The situation has greatly improved with the introduction of Symbolic model-checking methods which can standardly handle systems with up to 2^{150} states.

In addition to efficient representation of boolean assertions by the Ordered Binary Decision Diagrams (OBDD) data structure, symbolic methods call for set-based algorithms in which all the immediate successors of a given set of states can be computed in one step.

Milestone 3: The advent of Symbolic Model Checking techniques.

This caused the industry to sit up and take notice.

Principles of Symbolic Model-Checking



The assertion ψ_i characterizes all the states which are reachable from the initial condition by i or less steps.

$$
\begin{array}{rccl}
\psi_0 : & & & N_1 N_2\, y \\
\psi_1 : & \psi_0 & \vee & (N_1 T_2 \vee N_2 T_1)\, y \\
\psi_2 : & \psi_1 & \vee & T_1 T_2\, y \ \vee\ (C_1 N_2 \vee N_1 C_2)\, \overline{y} \\
\psi_3 : & \psi_2 & \vee & (T_1 N_2 \vee N_1 T_2)\, \overline{y} \\
\psi_\infty = \psi_3 : & \multicolumn{3}{l}{(N_1 \vee T_1)(N_2 \vee T_2)\, y \quad \vee} \\
 & \multicolumn{3}{r}{(C_1(N_2 \vee T_2)\ \vee\ (N_1 \vee T_1) C_2)\, \overline{y}}
\end{array}
$$

Deductive Verification

We may use the following rule to prove the invariance of assertion p:

Rule INV

I1.	$\Theta \rightarrow \varphi$
I2.	$\varphi \wedge \rho \rightarrow \varphi'$
I3.	$\varphi \rightarrow p$

$$\frac{\text{I1. } \Theta \rightarrow \varphi \quad \text{I2. } \varphi \wedge \rho \rightarrow \varphi' \quad \text{I3. } \varphi \rightarrow p}{\square\, p}$$

An assertion φ satisfying I1 and I2 is called inductive.

A Proof Strategy:

Find an inductive assertion φ strengthening p.

Sometimes We Can Take $p = \varphi$

Consider the following program for ensuring mutual exclusion by the use of semaphores.

$$y: \quad \textbf{natural where } y = 1$$

$$\left[\begin{array}{l} \ell_0 : \textbf{loop forever do} \\ \quad \left[\begin{array}{ll} \ell_1 : & \textbf{Non-Critical} \\ \ell_2 : & \textbf{request } y \\ \ell_3 : & \textbf{Critical} \\ \ell_4 : & \textbf{release } y \end{array}\right] \end{array}\right] \quad \| \quad \left[\begin{array}{l} m_0 : \textbf{loop forever do} \\ \quad \left[\begin{array}{ll} m_1 : & \textbf{Non-Critical} \\ m_2 : & \textbf{request } y \\ m_3 : & \textbf{Critical} \\ m_4 : & \textbf{release } y \end{array}\right] \end{array}\right]$$

The semaphore instructions **request** y and **release** y respectively stand for

$$\langle \textbf{await } y > 0\,;\ y := 0 \rangle \quad \text{and} \quad y := 1.$$

Wishing to verify the invariance of the assertion $y = 0 \ \lor\ y = 1$, we can take

$$\varphi \quad = \quad p \quad = \quad (y = 0 \ \lor\ y = 1).$$

Sometimes We Cannot

Wishing to verify the invariance of the assertion $\neg(at\_\ell_3 \ \wedge \ at\_m_3)$, guaranteeing mutual exclusion for the same program,

$$y: \quad \textbf{natural where } y = 1$$

$$\left[\begin{array}{l} \ell_0 : \textbf{loop forever do} \\ \quad \left[\begin{array}{ll} \ell_1 : & \textbf{Non-Critical} \\ \ell_2 : & \textbf{request } y \\ \ell_3 : & \textbf{Critical} \\ \ell_4 : & \textbf{release } y \end{array}\right] \end{array}\right] \quad \| \quad \left[\begin{array}{l} m_0 : \textbf{loop forever do} \\ \quad \left[\begin{array}{ll} m_1 : & \textbf{Non-Critical} \\ m_2 : & \textbf{request } y \\ m_3 : & \textbf{Critical} \\ m_4 : & \textbf{release } y \end{array}\right] \end{array}\right]$$

We must strengthen $p : \neg(at\_\ell_3 \ \wedge \ at\_m_3)$ into

$$\varphi: \quad \neg(at\_\ell_3 \wedge at\_m_3) \wedge ((at\_\ell_{3,4} \vee at\_m_{3,4}) \ \rightarrow \ y = 0)$$

or, equivalently,

$$y \geq 0 \ \wedge \ at\_\ell_{3,4} + at\_m_{3,4} + y = 1$$

A Strengthening Heuristic

Define the universal predecessor predicate transformer:

$$\rho \,\square\, \psi \quad = \quad \forall V' : \rho(V, V') \rightarrow \psi(V')$$

Obviously

$s \models \rho \,\square\, \psi$ iff all ρ-successors of s satisfy ψ.

Note that
I3. $\varphi \wedge \rho \rightarrow \varphi'$ is valid iff $\varphi \rightarrow (\rho \,\square\, \varphi)$ is,
and that
If φ is an invariant of $\mathcal{S}$ then so is $\rho \,\square\, \varphi$.

A (Top-Down) Strengthening Heuristic:

If φ violates I3: $\varphi \wedge \rho \rightarrow \varphi'$ then conjunct it with $\rho \,\square\, \varphi$.

Example: Deductive Verification

$$
\begin{array}{lll}
\textbf{in} & n & : \textbf{integer initially } n \geq 2 \\
\textbf{local} & y & : \textbf{integer initially } y = 1
\end{array}
$$

$$
\overset{n}{\underset{j=1}{\|}} \; P[j] :: \left[\begin{array}{l} \ell_0 : \textbf{loop forever do} \\ \quad \left[\begin{array}{ll} \ell_1 : & \textbf{Non-Critical} \\ \ell_2 : & \textbf{request } y \\ \ell_3 : & \textbf{Critical} \\ \ell_4 : & \textbf{release } y \end{array} \right] \end{array} \right]
$$

Mutual exclusion can be specified by

$$\varphi: \quad \left(\sum_{i=1}^{n} \chi(at\_\ell_3[i])\right) \quad \leq \quad 1$$

The corresponding inductive assertion can be given by

$$\psi: \quad y \geq 0 \quad \wedge \quad \left(\sum_{i=1}^{n} \chi(at\_\ell_{3,4}[i])\right) + y = 1$$

Establishing 5 preservation conditions plus $\psi \rightarrow \varphi$ verifies the system for any value of n.

Verification of Sequential Programs

Consider the bubble sort algorithm:

$$
\begin{array}{lll}
\textbf{in} & n & : \textbf{integer initially } n \geq 1 \\
\textbf{in-out} & A & : \textbf{array } [1..n] \\
\textbf{local} & i, j & : \textbf{integer}
\end{array}
$$

$$
\left[
\begin{array}{l}
\ell_0 : \quad i := n; \\
\ell_1 : \quad \textbf{while } i > 0 \textbf{ do} \\
\quad \left[
\begin{array}{l}
\ell_2 : \quad j := 1; \\
\ell_3 : \quad \textbf{while } j < i \textbf{ do} \\
\quad \left[
\begin{array}{l}
\ell_4 : \quad \textbf{if } A[j] > A[j+1] \textbf{ then} \\
\ell_5 : \qquad A[j] :=: A[j+1] \\
\ell_6 : \quad j := j + 1
\end{array}
\right] \\
\ell_7 : \quad i := i - 1
\end{array}
\right] \\
\quad \textbf{assert } \textit{sorted}(1, n)
\end{array}
\right]
$$

Where,

$$\textit{sorted}(u, v) \quad = \quad \forall k : (u \leq k < v) : A[k] \leq A[k+1]$$

Verifying Bubble Sort

$$
\begin{array}{lll}
\textbf{in} & n & : \textbf{integer initially } n \geq 1 \\
\textbf{in-out} & A & : \textbf{array } [1..n] \\
\textbf{local} & i, j & : \textbf{integer}
\end{array}
$$

$$
\left[
\begin{array}{l}
\ell_0 : \quad i := n; \\
\ell_1 : \quad \textbf{while } i > 0 \textbf{ do} \\
\ell_2 : \quad \textbf{assert } 0 < i \leq n \ \wedge \ \mathit{sorted}(i, n) \\
\qquad\qquad\qquad \wedge\ A[1..i] \leq A[i+1..n] \ \textbf{in} \\
\quad \left[
\begin{array}{l}
\ell_3 : \quad j := 1; \\
\ell_4 : \quad \textbf{while } j < i \textbf{ do} \\
\ell_5 : \quad \textbf{assert } 1 \leq j < i \\
\qquad\qquad\qquad \wedge\ A[1..j-1] \leq A[j] \ \textbf{in} \\
\quad \left[
\begin{array}{l}
\ell_6 : \quad \textbf{if } A[j] > A[j+1] \textbf{ then} \\
\ell_7 : \quad\quad A[j] :=: A[j+1] \\
\ell_8 : \quad j := j+1
\end{array}
\right] \\
\ell_9 : \quad i := i - 1
\end{array}
\right] \\
\ell_{10} : \quad \textbf{assert } \mathit{sorted}(1, n)
\end{array}
\right]
$$

The Evolution of Computational Models

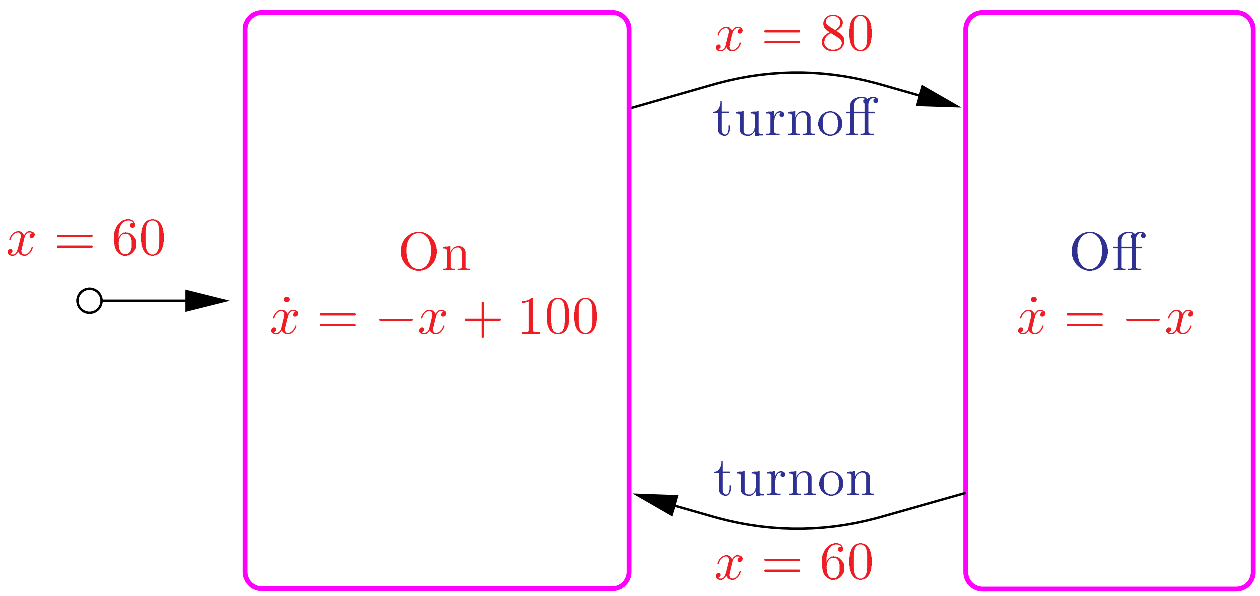
Reactive Systems
Expresses precedence relations
between events

Real Time
Can measure temporal distance

$$\Box\,(request \longrightarrow \Diamond_{\leq 5}\, response)$$

Hybrid Systems
Combination of discrete and Continuous
Components

An Example of a Hybrid System: A Thermostat



This junction of Computer Science and Control Theory proved highly beneficial for both disciplines.

So, How Close are We to Practical Formal Verification?

Very Close:

- FV is gaining wide acceptance in many areas:
  - Semi-conductor, ASICS, and other hardware industries are installing in-house research and application divisions for FV.
  - Communication Protocols are often formally verified.
  - In the area of safety-critical systems, FV will soon be mandated.

- Some companies (e.g. Intel) committed themselves to the use of FV methods as an emerging standard.

- Our ability to handle large systems (up to 10^{80} states) is continuously increasing. Some very large and intricate systems have been formally verified.

but, also

Still, Very Far:

- Handling systems with even 2^{350} states, covers a hardware design of only 350 gates. This is a far cry from modern chips holding up to $1,500,000$ gates.

- Formal verification is still carried out by experts who often employ ad-hoc methods which require creativity and ingenuity. It is an art rather than a science.

- The progress of FV in standard software development is still very slow.

In Conclusion: We have progressed a great deal, but there is still a long way ahead of us.

Major investments should be made in research and education.