The Algorithmic Verification of Reactive Systems

Pierre Wolper

Université de Liège

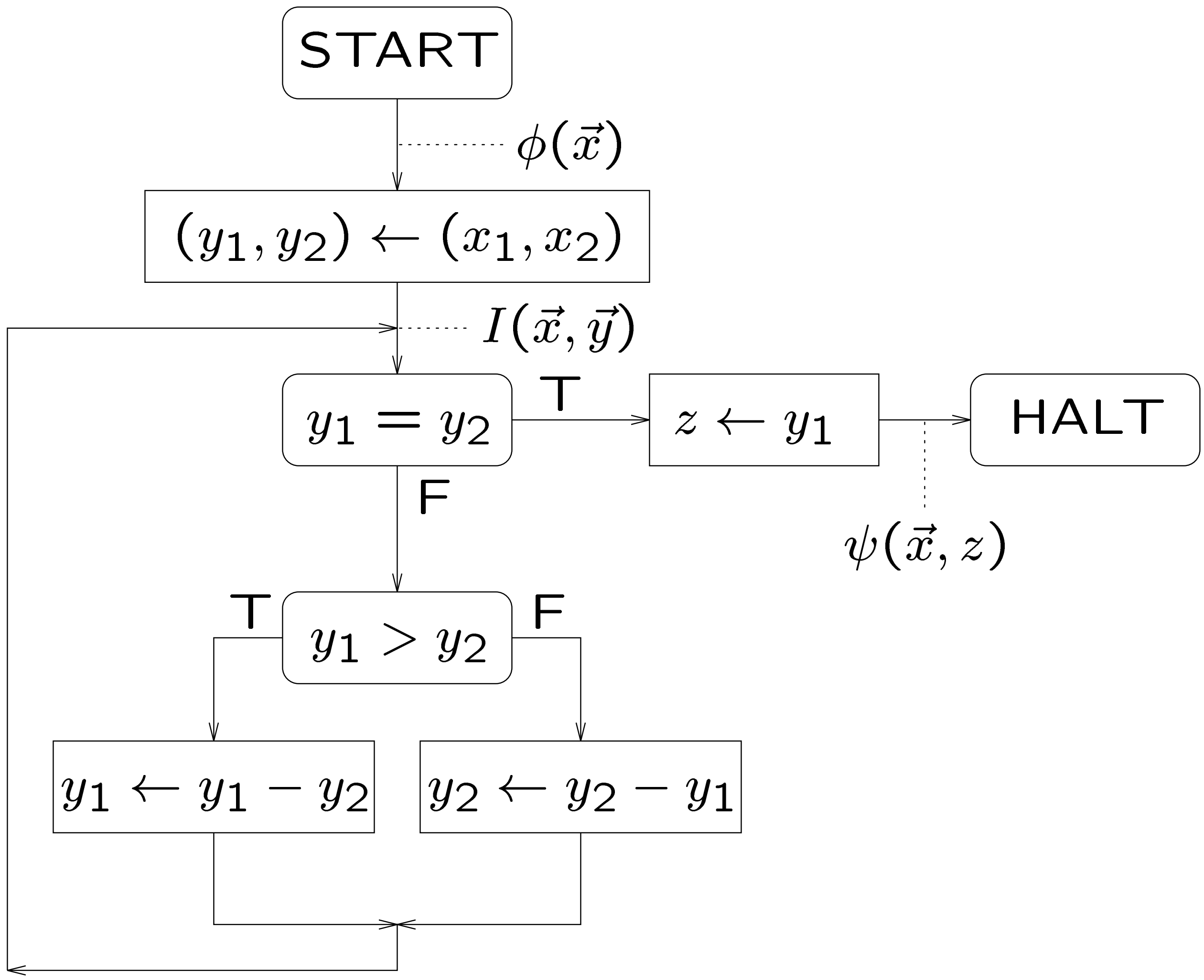
Inaugural Lecture

Verification:

Dreams and Reality

Verification:
How it all started

The most common example



What does this program compute?

What the program does

It computes the *greatest common divisor* (gcd) of (x_1, x_2), and you can prove it (Floyd 1967).

- Initially:

$$\phi(\vec{x}) : x_1 > 0 \wedge x_2 > 0$$

- At each iteration of the loop:

$$I(\vec{x}, \vec{y}) : x_1 > 0 \wedge x_2 > 0 \wedge y_1 > 0 \wedge y_2 > 0 \\ \wedge \, \mathrm{gcd}(y_1, y_2) = \mathrm{gcd}(x_1, x_2)$$

- When done:

$$\psi(\vec{x}, z) : z = \mathrm{gcd}(x_1, x_2)$$

One proves that I and ψ hold by induction on the length of computations. They are *invariants* and give properties of the states the program can reach.

Dream

We can get rid of errors in programs: just prove them correct!

Problems

- How do you find the adequate invariants?
- How do you reliably prove the inductive steps?
- What about termination?
- How do you handle programs that are not given as simple flowcharts?

Answer

Start a research program, in 10 years the problem will be solved.

Reality

(30 years later)

The overwhelming majority of programs are never proven correct. What went wrong?

Progress has been made, but falls short of what is needed.

Finding invariants

- Techniques exist, but their applicability is quite limited.

Proving inductive steps

- Theorem provers exist are quite powerful.
- Using them still requires a lot of effort and knowledge about logic and theorem proving.

Proving termination

- The mathematical framework for doing this is well known.
- It can sometimes be surprisingly hard to use.

Example

recursive function *threen* (n: *integer*):*integer*;
begin
if $(n = 1)$ **then** 1
  else if *even*(n) **then** *threen*$(n \div 2)$
    else *threen*$(3 \times n + 1)$;
end;

Dealing with real programming languages

- The basic idea came early: Hoare logic (1969).

A brief overview of Hoare logic

Hoare logic works with expressions of the form

$$\{\phi\}\ P\ \{\psi\}$$

where

- ϕ and ψ are assertions; P is program (or program fragment).
- The meaning of the expression is that if ϕ is true before executing P, then ψ is true after the execution of P.

Hoare logic then gives rules that establish the following.

- What expressions are true for basic instructions (assignments).
- How, knowing which expressions are true for program fragments P_1 and P_2, one can deduct the truth of expressions for programs built from P_1 and P_2:

$$P_1 \ ; \ P_2$$

$$\begin{array}{ll}\textbf{if } \mathit{test} & \textbf{then } P_1 \\ & \textbf{else } P_2\end{array}$$

$$\textbf{while } \mathit{test} \textbf{ do } P_1.$$

This is very simple and elegant, but runs into problems with more complicated programming constructs: procedures with parameters, exceptions and gotos, pointers, aliasing, complex data structures.

More Reality

Besides the technical problems, the dream of verification was based on a number of misconceptions.

- *Programs have relatively simple specifications:* true for the gcd, but think of a compiler, an operating system, or a word processor.
- *Verification is always needed:* bug-free software is clearly desirable in general, but there are many other ways of obtaining software that is sufficiently reliable for many applications (careful development methodologies, well designed testing, . . .)

- *Verification is worth the effort:* not if the effort is disproportionally large with respect to the rest of the software development effort.
- *Verification is a "noble" activity* (Dijkstra): it might be challenging for small intricate algorithms, but it is extremely tedious for large software packages

A task for the Mind
Or for the Computer?

The Mind

- Verification is an undecidable problem, it thus cannot be done with an algorithm.
- Verification (or verified design) is an intellectually pleasing activity that helps the programmer get a deep understanding of his programs.

The Computer

- The Mind shares the computational limits of the computer.
- Except for small clever programs, there is little intellectual pleasure. Furthermore, the issue is not intellectual pleasure by software development productivity.

- Manual verification is at least as likely to be wrong as the program itself.
- Verification is basically checking a design and in all other engineering disciplines this is done by computer. Why should software engineering be different?

Conclusion

Beyond all the other problems, the fascination with verification as an intellectual task has been detrimental to the field.

Is Verification a Dead Subject?

- As traditionally viewed, yes.
- But, interesting developments did occur during the last 30 years.
- It all started with concurrency.

The problem was to extend Hoare Logic to the concurrent composition of programs

$$P_1 \parallel P_2$$

The target rule was

$$\frac{\{p_1\}\ P_1\ \{q_1\} \qquad \{p_2\}\ P_2\ \{q_2\}}{\{p_1 \wedge p_2\}\ P_1 \parallel P_2\ \{q_1 \wedge q_2\}}$$

But, it is false if there is interaction between P_1 and P_2.

Starting with Owicki and Gries (1976), much effort was devoted to making this rule valid.

The Concurrent Composition Rule

Why does it fail?

- Concurrent processes can potentially interact (e.g. with shared variables) during the whole of their execution; not just at the beginning and the end.
- To know how concurrent program interact, it is not sufficient to know what happens at the beginning and at the end of their execution; what happens *during* the execution is also essential.
- So assertions should express properties of executions, not of initial and end states.
- A language for expressing properties of computations (sequences of states) is thus necessary.
- This is all the more the case since the global property of a concurrent program often cannot be given in terms of its input/output behavior.

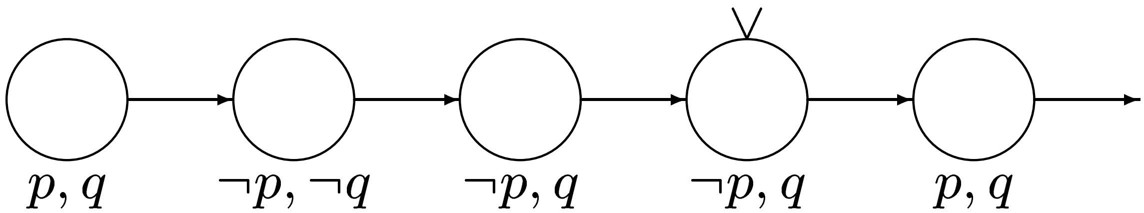
The Next Step: Temporal Logic

Temporal Logic (Pnueli, 1977) is now the most common language for describing computations or sequences of states. It consists of the following

- A logic for describing states which is
  - Propositional
  $$\neg p \wedge (q \vee r)$$
  - or first-order
  $$\forall x \exists y \; p(x, y).$$
- Operators that specify to which states of the sequence formulas should be applied:
  - No operator: the current state,
  - $\bigcirc$ (next): the next state,
  - $\Box$ (always): all future states,
  - $\Diamond$ (eventually): some future state,
  - U (until): all future states until a condition becomes true.

Examples

In the marked state of the sequence



the following formulas are *true*:

$$\neg p \wedge q$$

$$\bigcirc(p \wedge q)$$

the following formula is *false*:

$$\Box\, p$$

Why Temporal Logic?

- It is an easy to read extension of classical logic (propositional or first-order).
- It embodies a useful level of timing abstraction: an operator like $\diamond$ specifies a timing property in a qualitative way, not in a quantitative one.
- There are lots of nice results on temporal logic, for instance propositional temporal logic is decidable.

It was generally believed that a first-order logic was necessary for verification. The next leap was then to notice that *Interesting things can be done within propositional temporal logic* (1981).

Using Propositional Temporal Logic: Synthesis

- The decision procedure for temporal logic works by building a finite-state structure that generates the sequences that make the formula being considered true.
- Hence, if one views a propositional temporal logic formula as a specification, it is possible to *synthesize* algorithmically a finite-state program that satisfies this specification.
- The intended application was the synthesis of synchronization skeletons for concurrent programs (e.g. mutual exclusion protocols) which often are finite state.

The idea was nice, its failure is that it synthesizes a centralized solution rather than a distributed one.

Using Propositional Temporal Logic: Verification

Rather than synthesizing, one can verify (Queille & Sifakis, Clarke & Emerson)

- Start with a finite-state program and a temporal logic specification.
- Check that the finite-state program satisfies the temporal logic specification, which can be done algorithmically and quite efficiently.
- Since, in logical terms, one checks that the program is a model of the formulas, this approach is called *model checking*.

We thus have now a verification *algorithm*. Does it have applications?

The Target Application: Reactive Systems (Programs)

Reactive systems are systems that continuously interact with their environment.

Examples:

- parts of concurrent programs.
- interactive programs,
- process control software,
- communication software (protocols),

Why Reactive Programs?

Reactive programs are important and relatively easy to verify.

Important to verify?

- Very difficult to test (behavior might not be reproducible, exploitation context can be difficult to predict and model),
- Often appear in safety critical systems and their failure can be expensive (c.f. Ariane-5).

Easy to verify?

- Little data manipulation,
- Can be finite-state or have meaningful finite-state abstractions.

Thus our topic: the algorithmic verification of reactive systems.

Dream (1983)

All reactive systems can be verified algorithmically

Problems

- How does one obtain the specification? Can it always be written in propositional temporal logic?.
- How does one design easy to implement verification algorithms?
- The size of the finite-state representation of a concurrent program can be huge.
- Many reactive programs are not finite state.

Answer

Start a research program, in 10 years the problems will be solved.

Reality (1998)

Specifications

- Many extensions and improvements of temporal logic have been proposed.
- Good complete specification are still hard or impossible to obtain.
- However, this is not a real problem.
  - Checking simple (e.g., is state S reachable) or generic (e.g., is the program deadlock free) properties is already extremely useful.
  - The goal is no longer seen as proving that a system is completely, absolutely and undoubtedly correct,
  - The objective is to have tools that can help a developer find errors and gain confidence in his design. *This is achievable.*

Good Algorithms

- Much knowledge about temporal logic and model checking algorithms has been gained.
- Automata theory has been of great help.
- Good solutions do exist and have been implemented and used successfully.

The size of the state space

- This is a very central problem.
- Solutions exist, but there is no panacea.
  - Symbolic approaches: represent sets of states by boolean formulas and compute with these.

- Partial-Order methods: algorithmic means of avoiding the wasteful explosion in the number of states due to the modelling of concurrency by interleaving.
- Abstraction techniques: work with a simplified version of the system in a reliable way.

Dealing with Infinite state spaces

- This is a currently very active subjects.
- A number of approaches have been proposed but are not yet extensively tested.
- This line of work moves the topic closer to traditional verification.

Some Interesting Thoughts

- Exploring finite-state spaces started well before model checking, for instance for protocol analysis. One of the contributions of model checking is to attract much more effort and attention to the subject.
- With the problems of infinite state spaces, there is a move back towards more traditional verification techniques (invariants, theorem proving . . .), but the objectives are much more limited and realistic.
- The verification of hardware circuits is a booming application of model checking and related techniques. There are several reasons for this:
  - The finite-state assumption is not a problem for hardware.
  - The cost of errors can be enormous (c.f. the Pentium bug)
  - The complexity of designs is increasingly very rapidly.

Conclusions

- The algorithmic verification of reactive systems is a thriving topic.
- Several major companies are setting up research groups on the topic (Lucent, Intel, Cadence, . . .).