Modeling and Analysis of Timed Systems

Bengt Jonsson
Wang Yi
Uppsala University, Sweden

With contributions from
Tobias Amnell, Elena Fersman, Pavel Krcal, Leonid Mokrushin, Paul Pettersson

Trento, Feb. 2007

OUTLINE

- Untimed systems: modeling and specification
  - Motivation, Transition systems, Temporal logics,
  - Promela, Examples
- Untimed systems: model checking
  - Model checking algorithms, SPIN,
- Infinite-State and Parameterized Systems:
  - Model checking algorithms, decidability results,
- Timed Systems: modeling and theory
  - Modeling timed systems, Timed automata,
  - Verification of timed systems
- UPPAAL
  - Modeling, data stuctures & algorithms
- TIMES
  - From models to code "guaranteeing" timing constraints

Lecture 1

Introduction: Motivation and Sketch of Verification History

Why want bug-free programs?

- Testing consumes ~half of software development effort
- Several "expensive" accidents caused by bugs
  - Ariane 5 crash 1996
  - Pentium division bug
  - Mars pathfinder ceased to work 1997
  - Viruses, ….

Some of the Improvements needed

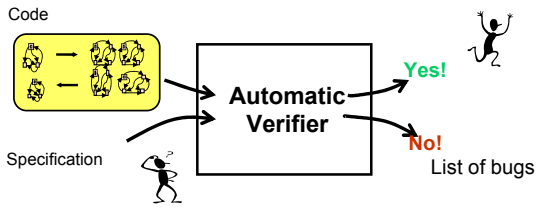
Better Development tools
- Programming languages
- development environments
- Libraries
- Software architectures

Better Skills
- Better designers
- better programmers
- Better testers and verifiers

Better Processes
- Better Collaboration
- better specifications,

Better Verification Techniques
- Testing and verification: Eliminating bugs

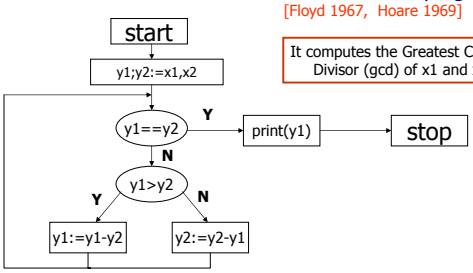
Dream: Program verifier



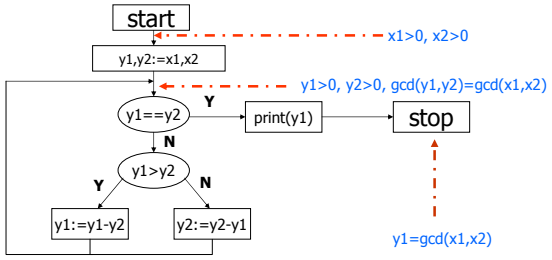
The dream started 40 years ago in 1960's
aiming at "bug-free software"

What does this program do?
[Floyd 1967, Hoare 1969]

It computes the Greatest Common Divisor (gcd) of x1 and x2



How can a program check this fact?



Can you check this **?**

Yes, you construct it manually
by finding the right loop invariant
And checking the appropriate conditins.

Yes, you construct it manually
by finding the right loop invariant
And checking the appropriate conditins.

Question: can you automate the proof ?

Software verification (now, a hot topic)

One more example (*Total correctness*)

```
Function foo(n)
begin
if n==1 then 1
        else if even(n) then foo(n/2)
                        else foo(3*n+1)
end
```

Does this program terminate for any n? (WCET?)

How are these techniques used today?

- This style of verification technique has been extended to concurrent, distributed, real-time ,... programs.
- It is a wide-spread tool for manual proofs,
- And for specifying procedures (pre-postconditions, contracts) in Eiffel, Java.
- It has been very difficult to automate
  - Difficult to find and prove invariants and variant functions
  - It to write complete specifications: what I really want?

History: Model-checking invented in 70's/80s

[Pnueli 77, Clarke et al 83, POPL83, Sifakis et al 82]

- Restrict attention to finite-state programs
  - Control skeleton + boolean (finite-domain) variables
  - Found in hardware design, communication protocols, process control
- Temporal logic specification of e.g., synchronization pattern
  - There are algorithms to check that Program satisfies: SPEC
  - e.g. Alternating Bit Protocol skeleton, around 140 states, 1984
- BDD-based symbolic technique [Bryant 86]
  - SMV 1990 Clarke, McMillan et al, state-space 10^{20}
  - Now powerful tools used in processor design
- On-the-fly enumerative technique [Holzman 89]
  - SPIN, COSPAN, CAESAR , KRONOS, UPPAAL etc
- SAT-based techniques [Clarke et al, McMillan, ...]

History: Model checking for real time systems, started in the 80s/90s

- Extension of model checking to consider time quantities
- Timed automata, timed process algebras [Alur&Dill 1990]
- Models, specfications, and algorithms can be extended
- KRONOS, Hytech, 1993-1995, IF 2000's
- TAB 1993, UPPAAL 1995, TIMES 2002

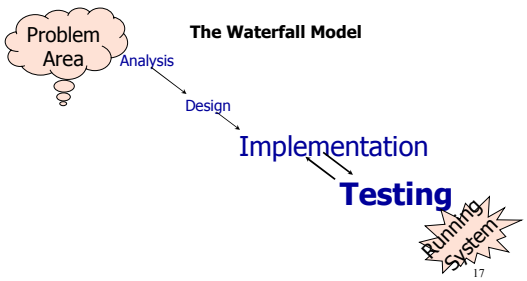
Merits of this simpler approach

- Checking simple properties (e.g. deadlock freeness) is already extremely useful!
- The goal is no longer seen as proving that a system is completely, absolutely and undoutedly correct (bug-free)
- The objective is to have tools that can help a developer find errors and gain confidence in her/his design. That is achievable
- Now widely used in hardware design, protocol design, and hopefully soon, embedded systems!

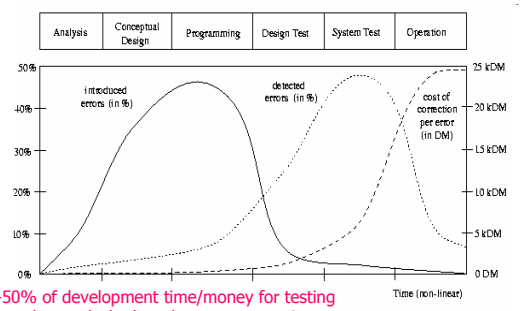
Why testing not good enough

- **Testing/simulation**: coverage problems, difficult to deal with non-determinism and concurrent computation
- **Formal verification/Model-Checking** (= **exhaustive testing** of software and hardware **design**) provides 100% coverage

Traditional software development

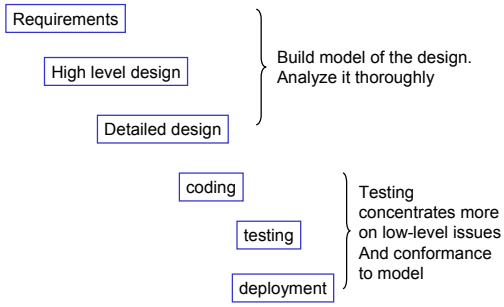


Introducing, Detecting and Correcting errors



- 30-50% of development time/money for testing
- Errors detected: the late the more expensive

Motivation: Purpose of Model Verification



Some Verification and Validation Techniques

Testing: Needs code. Incomplete

Static Program Analysis: Needs code. Can systematically find certain classes of errors.

Code Reviews: Also needs code. Can be time-consuming

Design Reviews: Does not need code. Not automated

Prototyping and Simulation: Needs good tool support. Incomplete

Model Checking:

- Can be done early in the design cycle.
- Automated (provided tools available)
- Systematic, can be made rather complete
- A model must be constructed (at a suitable level of abstraction)

Problems that can be addresed by Model Checking

Checking correctness of

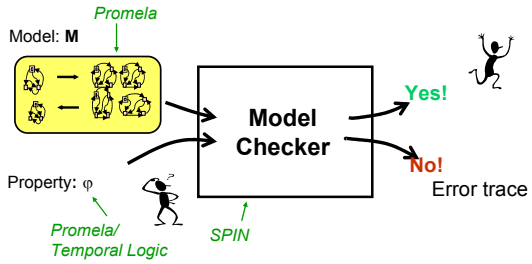
- Communication protocols
- Distributed Algorithms
- Controllers
- Hardware circuits
- Parallel and distributed software
- Embedded and real-time systems and software

e.g.,

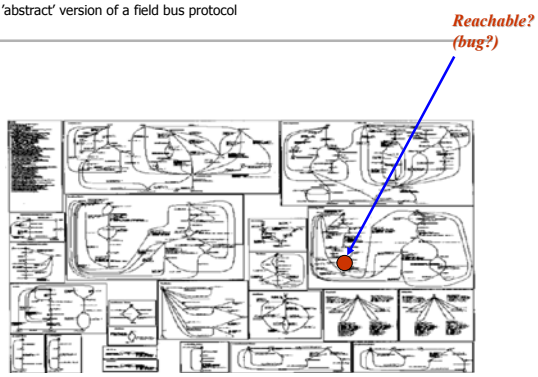
Absence of race conditions, proper synchronization, ...

Model checking is the appropriate technique when there are many many different scenarios of interaction between components in a system

Overview of Model Checking



An 'abstract' version of a field bus protocol



Remaining Problems

Constructing a Model

- Not so easy, this course will make you experts

Checking that the actual system/software conforms to the model

- Hard problem: there are several techniques:
  - Conformance testing
  - Static program analysis
  - Automated code generation

Model-Checking may complement testing to find (design) Bugs as early as possible