Checking Models, Proving Programs, and Testing systems

Marie-Claude Gaudel

LRI, Université de Paris-Sud & CNRS

Outline of the talk

- Some hopefully “clear” definitions
  - Models, Programs, Systems, Properties
  - Model-checking, Program proving, Testing
    - Brief state-of-the-art
- Not so clear variants of the definitions above
  - Run-time verification, Model-checking programs, Coverage in model-checking, Bounded model-checking, Model-based testing, Program checking, Proof approximation…
- Along the talk: some examples of cross fertilisation

Some “clear” definitions

- **Models**
- Programs
- Systems
- Properties

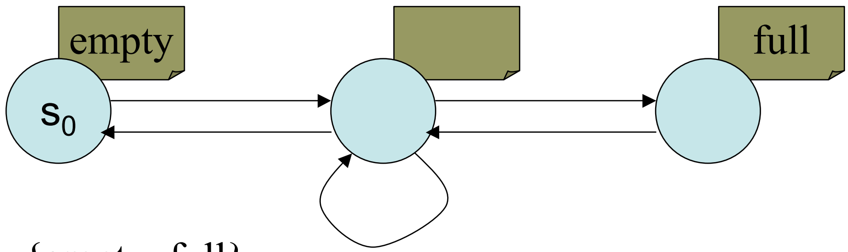
- Model-checking
- Program proving
- Testing

Models: an heavily overloaded* term

- Here models – as they are used for model-checking – are just *annotated graphs*
  - A finite set of states, S
  - Some initial state, s_0
  - A transition relation between states, $T \subseteq S \times S$
  - A finite set of atomic propositions, AP
  - A labelling function $L : S \rightarrow \mathcal{P}(AP)$
- Richer similar notions:
  - Labelled Transition systems, LTS
  - Finite State machines, FSM
  - State charts, …

** For a physicist a “model” is a differential equation; For a biologist, it may be … mice or frogs*

An example



AP = {empty, full}

Some LTL formula that are valid for this model:

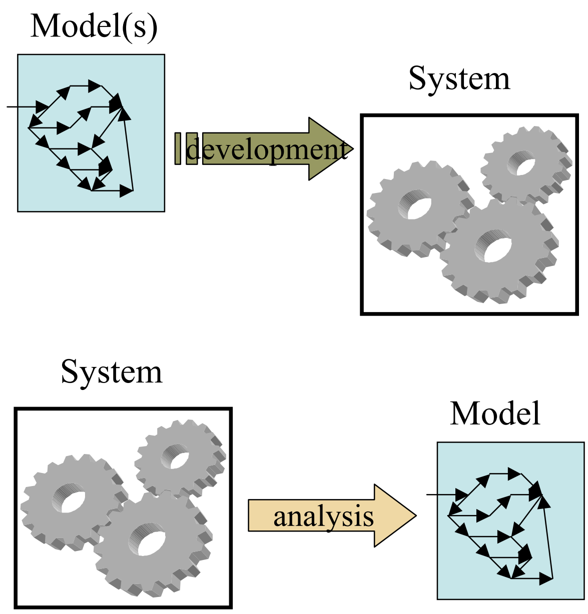
empty ⇒ (X ¬empty)

full ⇒ (X ¬full)

(X is for neXt)

What are models good for?

- System description and design:
  - The future system must conform to the model(s)
  - The model(s) may be used as a starting point for (automatic) development
- System analysis
  - Observing the existing system, one extracts a model and studies it
- …
- Essential role in V and V and quality assurance



Some “clear” definitions

- Models
- **Programs**
- Systems
- Properties

- Model-checking
- Program proving
- Testing

Programs

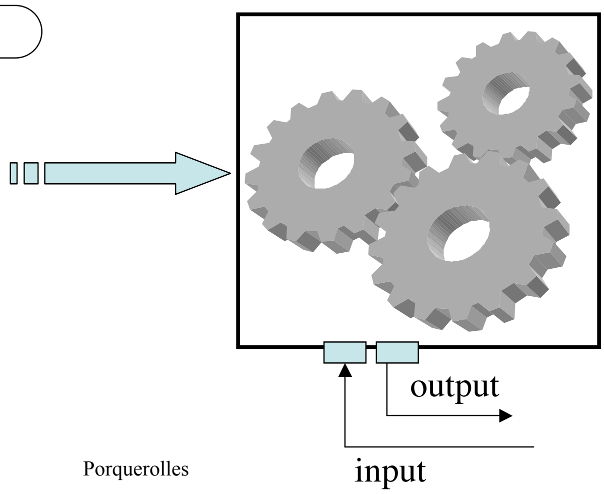
- Everybody knows what it is ☺
- Here:
  - A program is a piece of text in a (hopefully) well defined language
  - There is a syntax, some semantics, and **compilers**
- “A program is a very detailed solution to a much more abstract problem” [Ball, 2005]

```
{i=0 ; read(x);
repeat {
i++ ;
perd(x, i); }
until  term(i,x) ;
…
```

Why are programs useful?

- They can be compiled and embedded into some system

```
{i=0 ; read(x);
repeat {
i++ ;
perd(x, i); }
until  term(i,x) ;
…
```

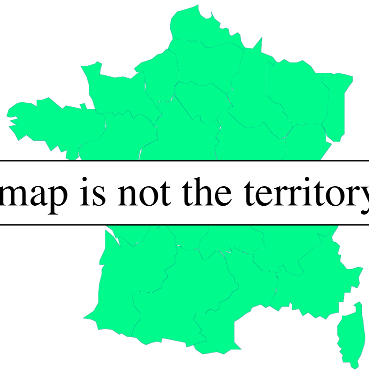


Interlude

```
{i=0 ; read(x);
repeat {
i++ ;
perd(x, i); }
until  term(i,x) ;
…
```

CORRECT!

Interlude (cont)



"A map is not the territory"

- A program text, or a specification text, is not the system

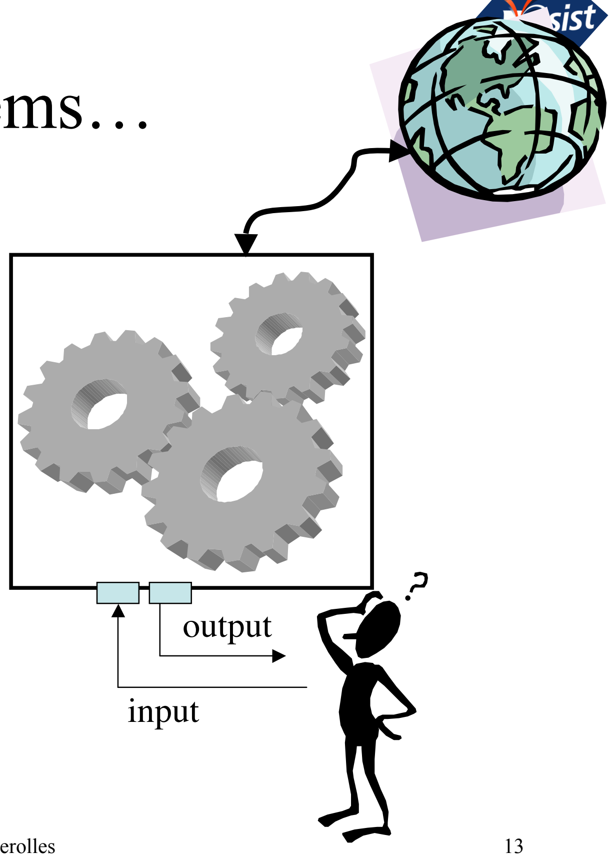
Some "clear" definitions

- Models
- Programs
- **Systems**
- Properties

- Model-checking
- Proof
- Testing

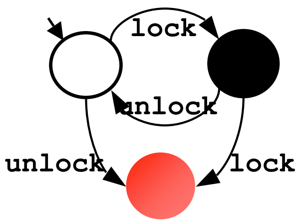
Systems…

- A system is a dynamic entity, embedded in the physical world
- It is observable via some limited interface/procedure
- It is not always controllable
- Quite different from a piece of text (formula, program) or a diagram

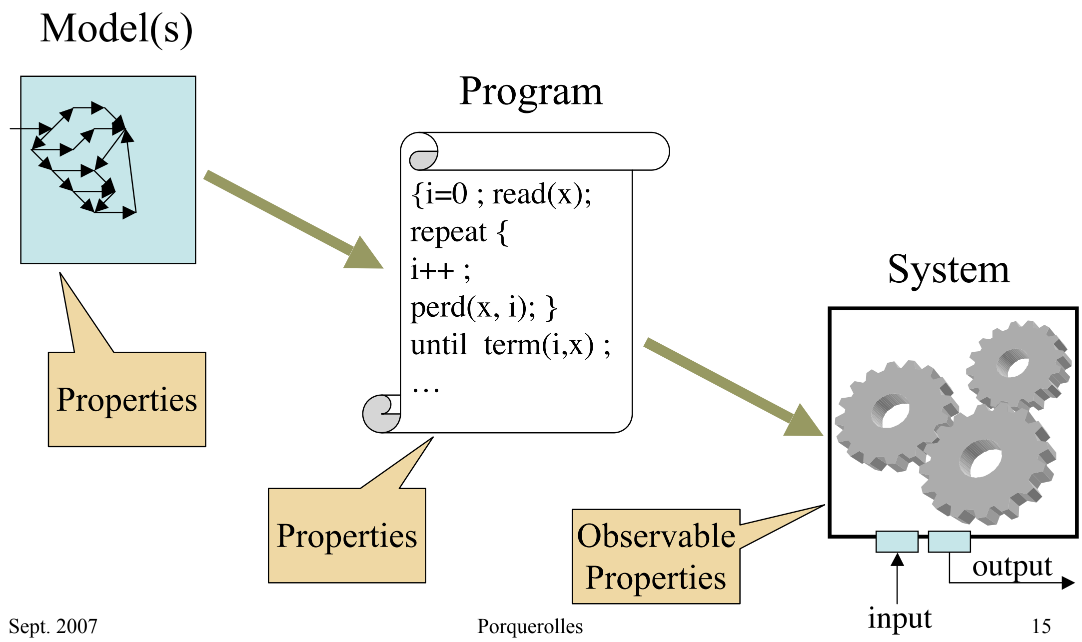


Systems are the actual objects of interest

- How to ensure that a system satisfies certain properties?
- Properties?
  - Texts in natural languages…
    - "Calls to **lock** and **unlock** must **alternate**."
  - Formulas in a given specification logic
    - (locked ⇒ X unlocked) ∧ (unlocked ⇒ X locked)
  - Sets of mandatory or forbidden behaviours



The Classical Process…



Some “clear” definitions

- Models
- Programs
- Systems
- **Properties**

- Model-checking
- Program proving
- Testing

Properties…, Specification Languages…

- Logic-based specification languages
  - VDM, Z, CASL, HOL, B, **JML**, …
  - Temporal Logics: **LTL,** CTL, …
- Behaviour-based specification languages
  - Lotos, Promela, CSP, State charts, Petri Nets, Timed automata…
- Usages:
  - Global requirement on the system as a whole, or of some subsystems
  - Assertions in programs and models: pre-conditions, post-conditions, invariants

Example: some JML invariant

```
public /*@ pure @*/ class ModelSet {
    /*@ public invariant (\forall Object e1, e2;
        @  this.add(e1).has(e1)
        @  && this.add(e1).add(e2).equals(this.add(e2).add(e1))
        @  && this.add(e1).add(e1).equals(this.add(e1))
        @  && (this.equals(new ModelSet()) ==> !this.has(e1)) )
        @*/
    public ModelSet() {…}
    public boolean has(Object o) {…}
    public ModelSet add(Object o) {…}
    public boolean equals(/*@ nullable @*/ Object o) {…}
}
```

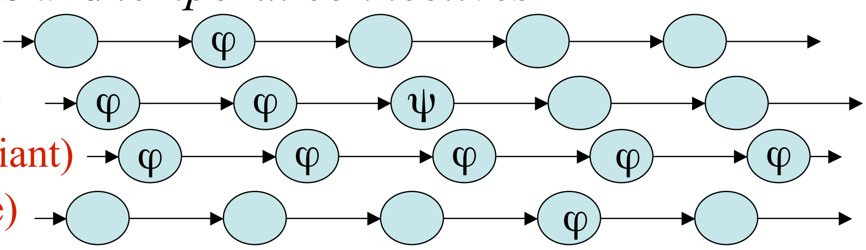
Example: JML post-conditions

```
public /*@ pure @*/ interface UModelSet {
    public boolean has(Object o) ;

    /*@ ensures \result.has(o) &&
       @ (\forall Object el; el != o ==> this.has(el) == \result.has(el)) ;
    @*/
    public UModelSet add(Object o) ;

    /*@ ensures (\forall Object el; ! \result.has(el)) ;
    public UModelSet emptySet() ;
}
```

Sorry: in JML the post-conditions are above the concerned method ☹

Example of temporal Logic : quick introduction to LTL

- Syntax: LTL formulas are built from a set AP of *atomic propositions* and are closed under Boolean connectives and *temporal connectives*
  - X (next)
  - U (until)
  - G (invariant)
  - F (future)



- Semantics
  - Given a finite model M
  - M *satisfies* a LTL formula φ if *all traces of M satisfy φ*

LTL (cont.)

- Example
  - *G(¬request ∨ (request U grant))*
  - "whenever a request is made it holds continuously until it is eventually granted"
- Interest of LTL
  - Checking whether a finite model M satisfies a LTL formula φ can be done
  - in time $O(|M|.2^{O(|\varphi|)})$
  - in space $O((|\varphi| + \log|M|)^2)$

Linear in the size of the model

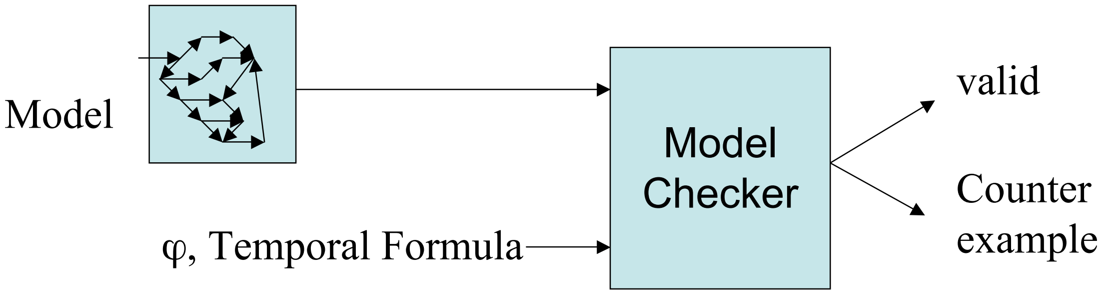
- Cons: it's often difficult to express realistic properties => CTL (quantifications on traces) & others, but … tricky anyway

Some "clear" definitions

- Models
- Programs
- Systems
- Properties

- Model-checking
  - Concise state-of-the-art
- Program proving
- Testing

Model-Checking

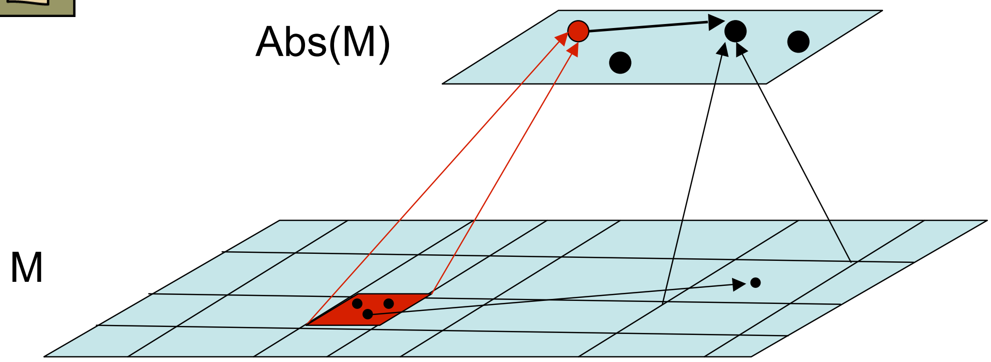


Algorithmic approach: exhaustive exploration of the model
A well-known example: SPIN, where models are described in Promela and checked against LTL formulas
Big issue: size of the model (esp. due to concurrency). Huge models are attainable… but it is not enough

Model-Checking

- The state-of-the art in a few words : **struggle against size, …and infinity**
- Symbolic model-checking: BDD (set of states) and fix-point operators
  - SMV: hundreds of boolean variables (CTL), more than 10^{20} states, 10 years ago
- SAT-based model-checking and bounded model-checking
- Abstraction, and CEGAR « counter example-guided abstraction refinement »
- Partial-order reduction (in case of concurrency)

Abstraction of a model



Behaviour preservation:
$<s, s'> \in T_M$ iff $<Abs(s), Abs(s')> \in T_{Abs(M)}$

Abstraction makes it possible to

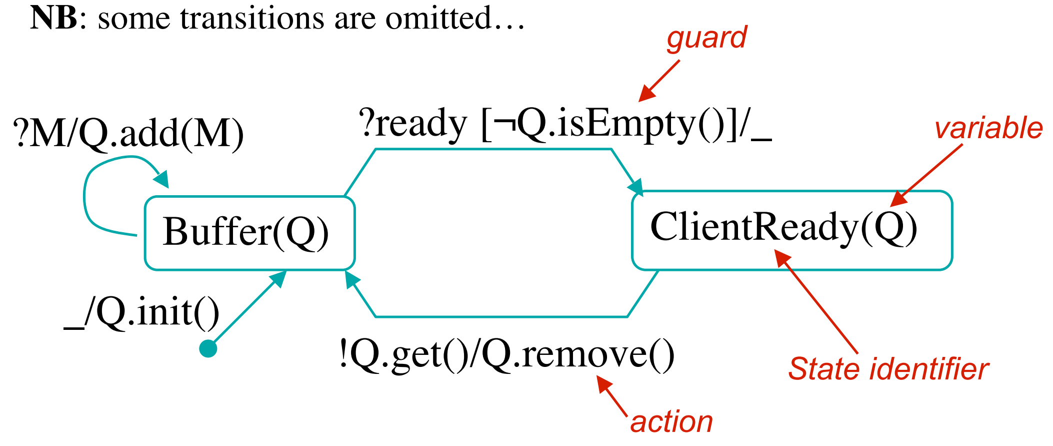
- dramatically reduce big models
- specify and analyse infinite models

Why infinite models?

- Underlying models of several specification notations:
  - Lotos, SDL, Promela, CSP with value passing mechanisms, UML statecharts…
- Underlying models of *programs*
- Notation for infinite models:
  - *State identifiers* are decorated with typed variables
    - they denote classes of states, possibly infinite
  - *Transitions* between such classes of states are labelled by events, guards, and actions
    - where variables may occur
    - Where actions may modify variables values
  - They denote classes of transitions, possibly infinite

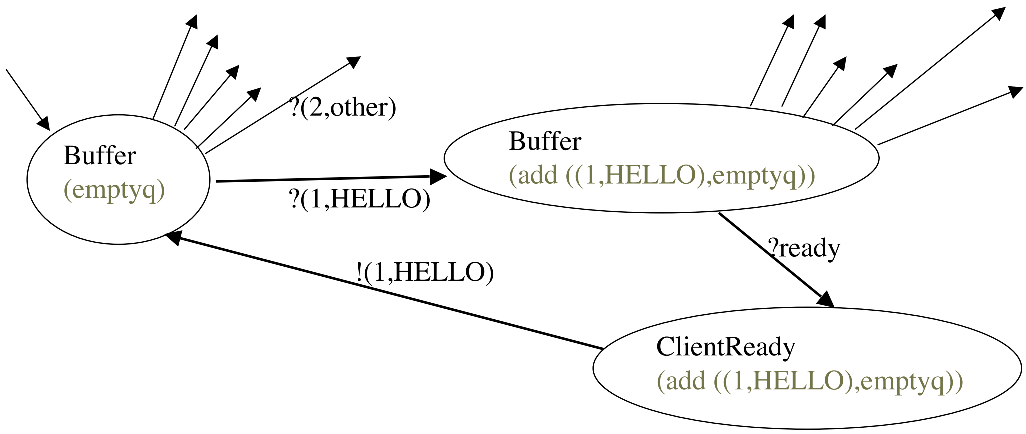
An example: buffer with priority a finite description

NB: some transitions are omitted…



M: Message, couples of text and priority
Q: Queue of Messages, with init, add, remove and get operations

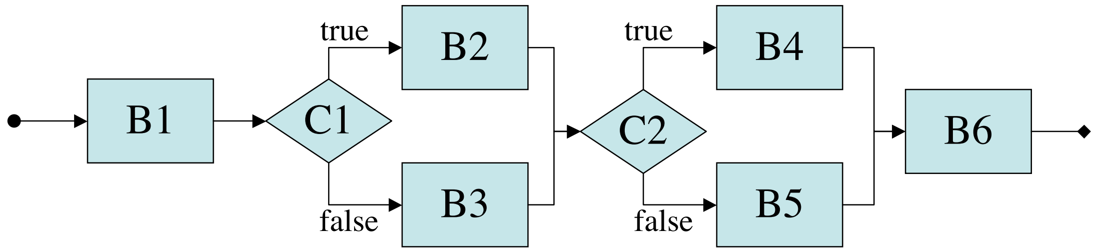
A very small part of the underlying model



Big issue: reachability of states and transitions…it is not decidable ☹
The finite description is an over- approximation of the infinite model

Unfeasible traces

- It is a classical problem in structural testing



- C1 and C2 may be incompatible
  - more precisely: $C1_{after\ B1} \wedge C2_{after\ B1.B2}$ may be unsatisfiable => B1 C1 B2 C2 B4 B6 is not feasible

A few model-checkers

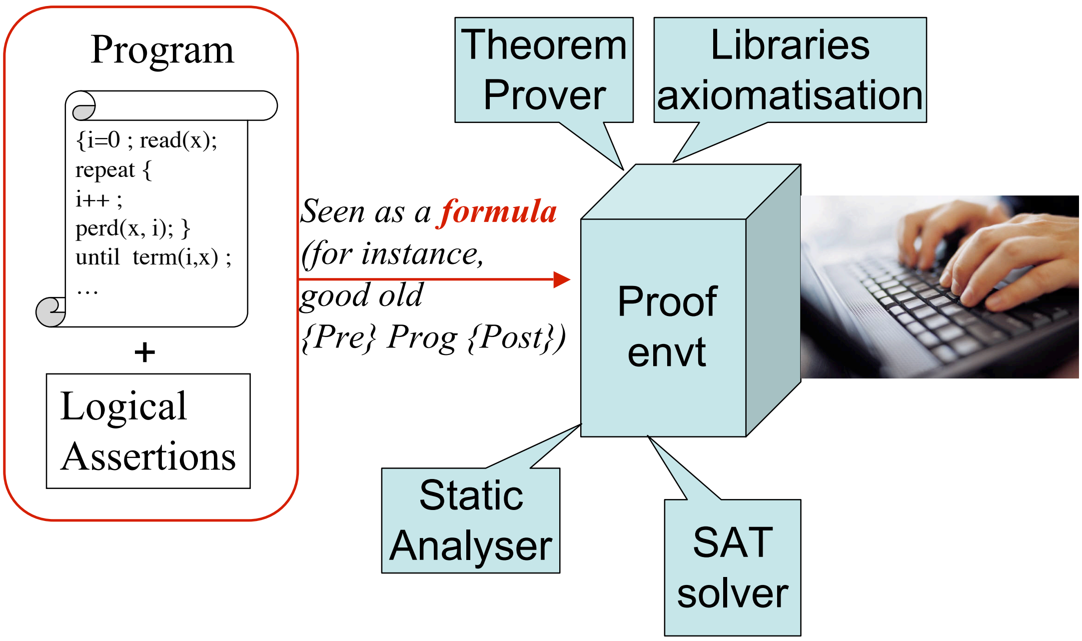
- SPIN (Promela, *LTL*)
- NuSMV 2 (*CTL*) combines BDD-based model checking with SAT-based model checking.
- FDR (CSP, *refinements*)
- *Timed automata*: UPPAAL, KRONOS
- *Stochastic models*: PRISM, APMC

Some “clear” definitions

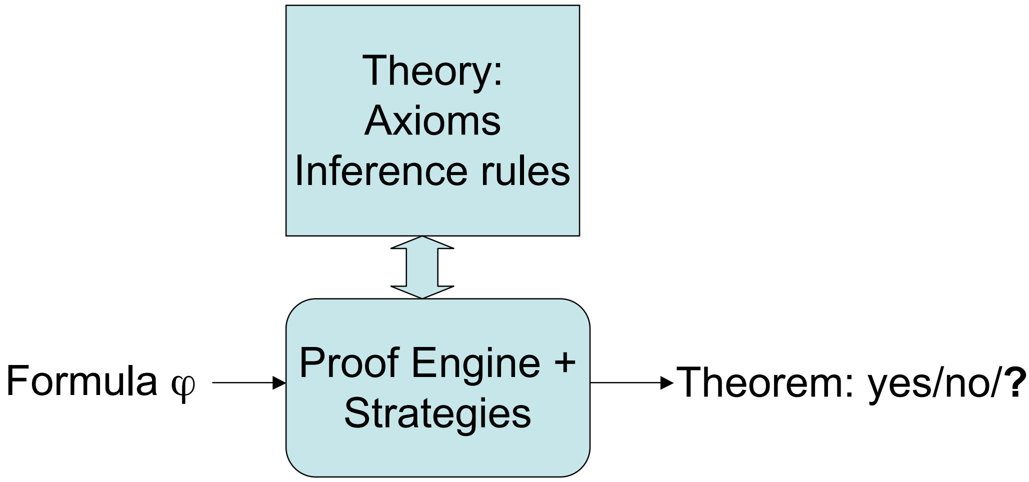
- Models
- Programs
- Systems
- Properties

- Model-checking
- Program Proving
  - On-going progresses
  - Static Analysis
- Testing

Program Proving



What is a proof?



Syntactic process: *transformation of φ, via the inference rules, into some axioms*

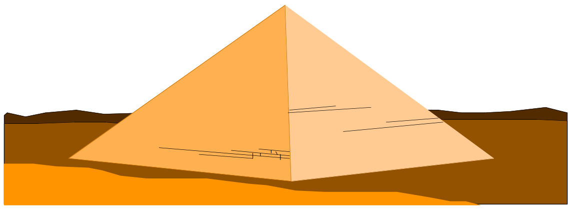
Not automated for powerful theories (f. i. inductive ones)

Program Proving

- Significant and continuous progresses
  - Great theorem provers: Coq, Simplify, HOL/Isabelle, PVS…
  - Powerful static analysis techniques
- Tendency
  - Environments specialised for given couples <programming language, specification/assertion language> : Java/JML, C#/Spec#
    - The assertion language is tailored for the programming language
  - Libraries of abstract modelling types (collections, etc)
  - Big industrial investments : HP, Microsoft Research, …

A personal remark

Good old ideas (Hoare's logics, Dijkstra's wp calculus) are still basic.



But now, in addition…

Progresses and challenges

- *Side-effects and aliasing* handled by various program logics
  - Reasoning about heap structures and aliasing ☺, but… pb with invariants of complex object structures ☹
- Reasoning on *breaking out of loops*, or *catching exceptions* solved by "logics for abrupt termination" ☺
- *Dynamic method binding* and *inheritance* partially handled by "behavioural subtyping" 😐
- Gap between some abstract modelling types and concrete types (*quantifications, _.equals() versus =*) ☹
- *Non-termination* (loop variants, model-checkers) handled in various cases 😐

[Leavens, Leino, Muller, FAC 2007]

Advances in static analysis

- Static analysis provides ways to obtain information about possible executions of a program without running it.
- It is an approximation
  - indecidability of feasability => a super-set of the actual executions is considered => possibility of false alarms or inconclusive answers
- Main approaches:
  - Abstract interpretation [Cousot 77] (f.i. the ASTRÉE tool)
  - …Model-checking (sometimes called Software model-checking, see later)

The static analyser ASTRÉE

- Structured C programs, without dynamic memory allocation and recursion, with no side-effect
- Check that some kinds of "run-time errors" cannot occur during any execution in any environment
  - Division by zero, Out of bound array indexing
  - Arithmetic overflow
  - User-defined assertions
- "Domain-aware" (logic and functional properties of control/command theory)
  - "Miracles" on the considered family of programs and properties

Recommended reading on program proof and static analysis

- *Verified Software: Theories, Tools, Experiments*
- Conference in Zurich, fall 2005
- Under the auspices of Tony Hoare's grand challenge: « Verifying Compiler »
- http://vstte.ethz.ch

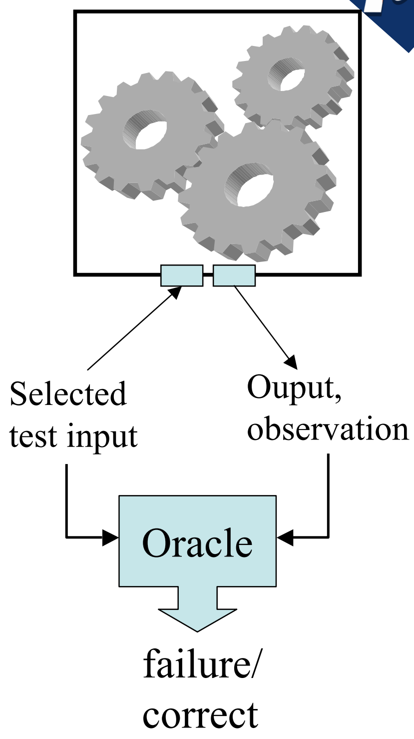
Some "clear" definitions

- Models
- Programs
- Systems
- Properties

- Model-checking
- Program Proving
- Testing
  - Tendencies, progresses

Testing

- The actual system is executed for a finite set of selected inputs
  - NB: selected test sequences for reactive systems
- These executions are observed, and a decision is made on their conformance w. r. t. some specification
- Issues :
  - Selection
  - Oracle
  - Control, non-determinism
  - Assessment of the result of a test campaign



Selection

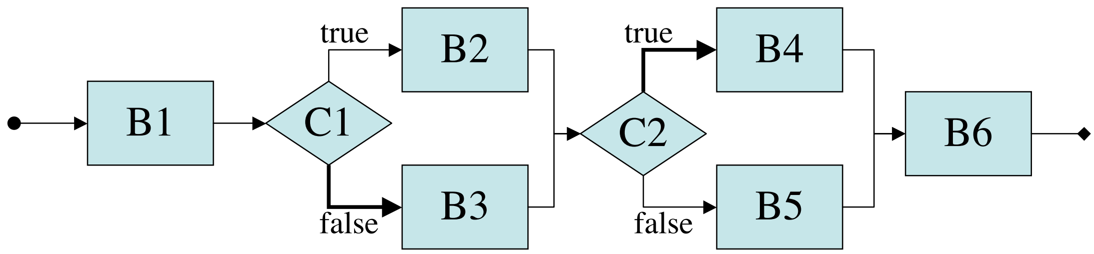
- Infinite input domain → finite test set, likely to lead to as many failures as possible
- The selection process can be based on:
  - Some characteristics of the input domain
  - The structure of the system or of the program
  - Some specification/model of the system, and or its environment
  - Test purposes
- Coverage criteria of … the input domain, the structure of the system or of the program, the specification or the model are very popular
- Actually, the general idea is
  - Infinite input domain → finite number of test cases (input sub-domains) that correspond to uniform behaviours w. r. t. failures
  - *Uniformity hypothesis, regularity hypotheses*

More on Selection Hypotheses

- They formalise Common Test Strategies
  - **Uniformity hypotheses:** based on some partition of the input domain. "passing one test in each uniformity sub-domain is sufficient"
  - **Regularity hypotheses:** based on some size function on the tests. *"If all the tests of size less than or equal to a given limit are passed, then it is true for all the Input Domain"*
- Possibility to derive them from
  - **Static analysis** of the program, or symbolic evaluation (and to prove them using program proving)
  - Analysis of some specification/model, and to prove/check them
- The notion of uniformity sub-domain is similar to abstraction, *but it is not clear that the same abstractions must be used for testing and model-checking*

From test cases to test inputs

- Back to good old structural testing



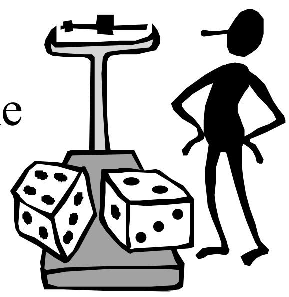
- The test case corresponding to executions of path B1 C1 B3 C2 B4 B6 is the path predicate $\neg C1_{after\ B1} \wedge C2_{after\ B1.B2}$
  - This constraint must be solved to get some test input
  - NB: may be unsatisfiable

Constraint solvers

- Essential tools for test generation (and theorem proving)
- Better and better systems for
  - SAT-solving
  - Finite domains (f. i. boolean constraints)
  - Linear arithmetics
  - Specific domains (f. i. finite sets)
- In more general cases improvements due to
  - Randomisation of the search of a solution
  - Approximation (± abstract interpretation)
- Not yet powerful enough for the needs of realistic system testing…

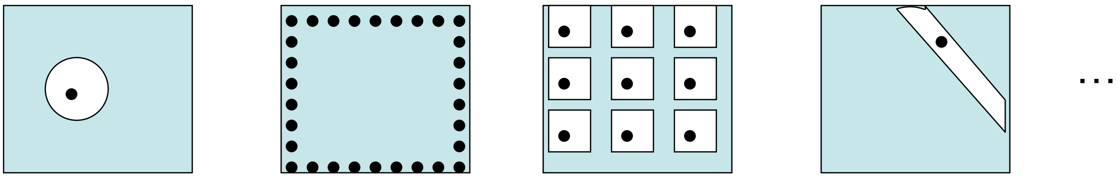
Random Testing

- These methods can be classified into three categories :
- those based on the input domain
  - Adaptive random testing
  - Stochastic optimisation (simulated annealing, genetic algorithms)
- those based on the environment
- and those based on some knowledge of the behaviour of the IUT
  - Random walks
  - Coverage-biased random selection



A few words on adaptive random testing

- Failure-causing inputs tend to cluster together following some patterns (see naive examples below)
- These patterns are used to define probability distributions on the input domain
  - *Statically* (previous knowledge on the type of system) or *dynamically* (± random walks, stochastic optimisation, learning)



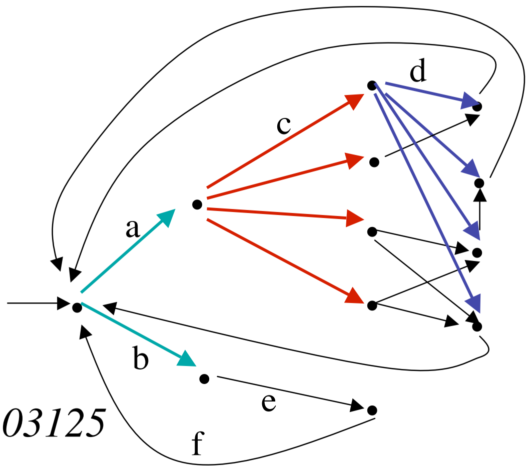
A few words on coverage-biased random selection

- Old classical idea for simulation and testing: *random walks*
- A random walk in the state space of a model (a control graph, etc) is:

  a sequence of states s_0, s_1, …, s_n such that s_i is chosen uniformly at random among the successors of the state s_{i-1},
- It is easy to implement and it only requires local knowledge of the graph.
- Numerous applications in
  - Testing (protocols), simulation
  - Model-checking (recent works)

Drawback of classical random walks

The resulting coverage is dependent on the topology…



Classical random walks, length 3:

$Pr(a; c; d) = 0.5 \times 0.25 \times 0.25 = 0.03125$

$Pr(b; e; f) = 0.5$

Uniform random sampling of traces, length 3:

$Pr(a; c; d) = Pr(b; e; f) = 0.1$

Uniform generation of bounded paths in a graph

- Counting [Flajolet et al.]: Given any vertex v, let $l_v(k)$ be *the number of paths of length k that start from v*
  - we are on vertex v with m successors $v_1, v_2, \ldots, v_m$
  - condition for path uniformity: choose v_i with probability $\boldsymbol{l_{vi}(k-1)/l_v(k)}$
- Application to various criteria based on paths
- Generalisation to node coverage, branch coverage
- Assessment of the quality of the coverage ☺
- Application to C programs (AuGuSTe) and to models
- The RASTA group, LRI, [ISSRE 2004], [Random Testing Workshop 2006], [Random Testing Workshop 2007]

Related work

- NB: in any case, constraint solving is required
  - Open issue: uniform constraint solving (work in progress : [Gotlieb, IRISA, 2006])
- Some similar tools
  - PathCrawler, [Nicky Williams, CEA]
  - DART, [Godefroid & al., PLDI 2005], linear constraints
  - In both cases “dynamic” test generation
  - Only considered criteria: all paths $\leq$ given length, no other coverage criteria

You are all invited!

Second International Workshop on
RANDOM TESTING (RT 2007)

co-located with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)

Atlanta, Georgia, USA, November 6, 2007

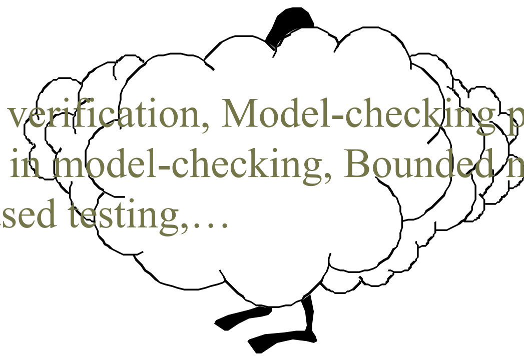
http://www.mathematik.uni-ulm.de/sai/mayer/rt07/

Ask for the program!

Outline of the talk

- Some “clear” definitions
  - Models, Programs, Systems, Properties
  - Model-checking, Proof, Testing
- Not so clear variants of the definitions above
  - Run-time verification, Model-checking programs, Coverage in model-checking, Bounded model-checking, Model-based testing,…
- Along the talk: some examples of cross fertilisation

? ? ?

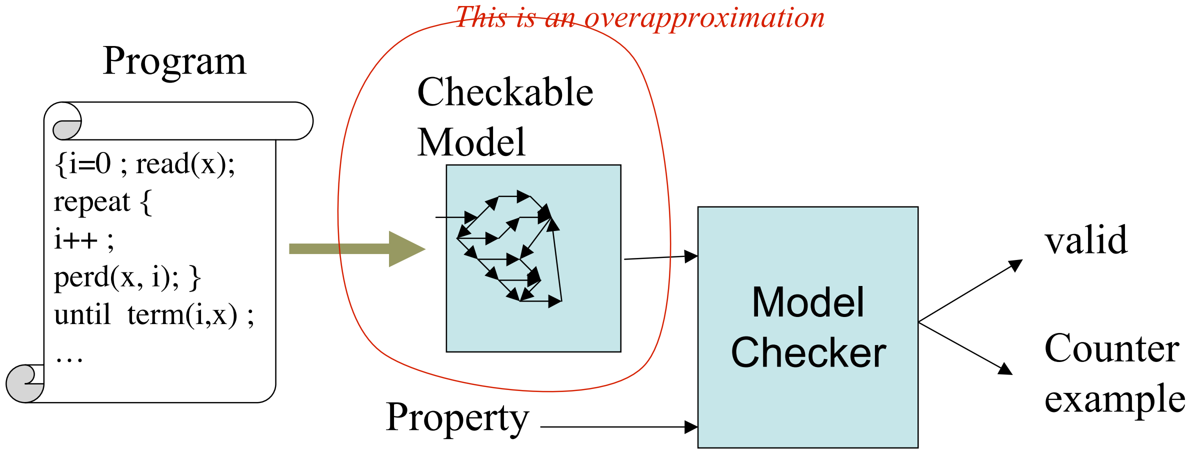


Run-time verification, Model-checking programs, Coverage in model-checking, Bounded model-checking, Model-based testing,…

Zoom in on

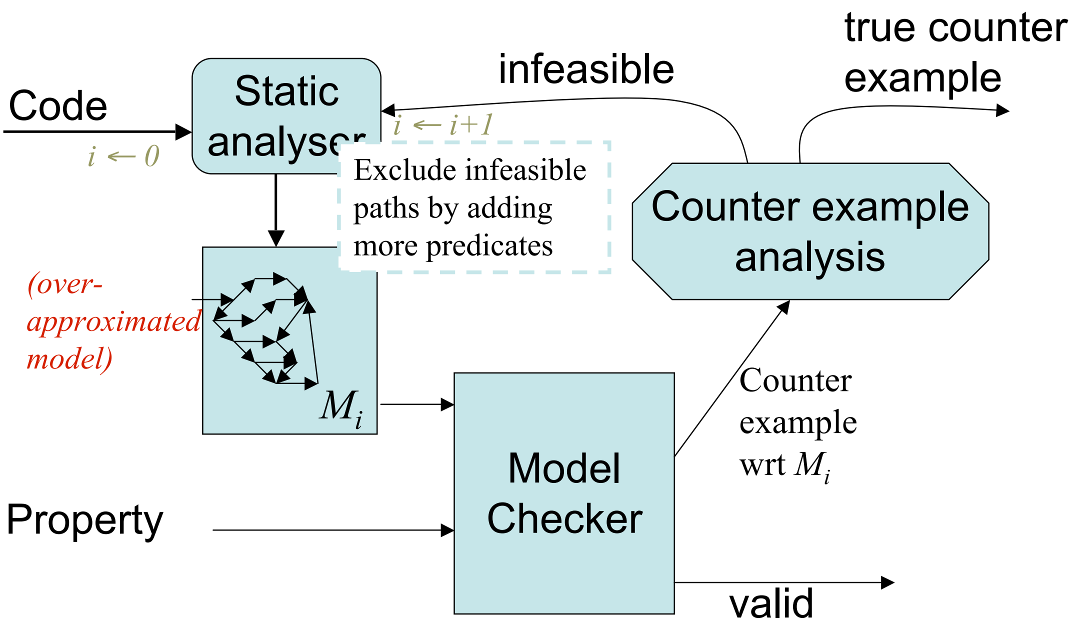
- What people call Software Model Checking
- What people call Model Based Testing
- What is Test Generation using Model-checking

Software model-checking



More precisely… CEGAR

« counter-examples guided abstraction refinement »



An example: SLAM/SDV

- Specific to Windows device drivers
- Reverse engineers a Boolean program from the C code, that represents how the code uses the driver API
- SLAM uses symbolic model-checking (SDV) and abstraction refinement to check that the driver properly uses the driver API
- SDV includes some important domain expertise: a set of API usage rules
- Tailored for a certain class of errors
- Up to 7000 lines C drivers, successful

[Ball et al., Microsoft research]

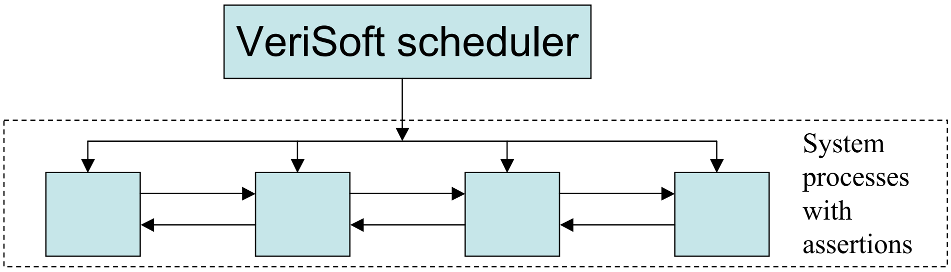
Another example: VeriSoft

[Godefroid et al., Lucent Technologies]

- Deals directly with concurrent systems written in C or C++
- Complexity of states => “state-less” search, based on transition sequences
- Partial-Order reduction => reduction of the number of paths, “selective” DFS search and *no explicit construction of the model*
- Search for deadlocks, assertion violations, bounded divergence and livelock
- *Dynamic observation* of the concurrent system

What is VeriSoft ?

Rather a scheduler/simulator than a model-checker



- automatic state-space exploration mode
- or interactive simulation mode
- based on
  - model-checking algorithms
  - dynamic analysis of independence of transitions

More on VeriSoft

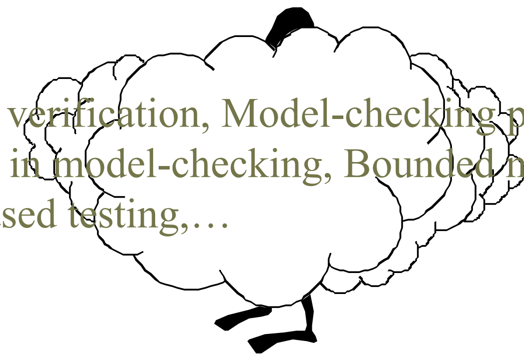
- Dynamic pruning
  - At each state reached during the search, VeriSoft computes a *subset of the enabled transitions* and ignores the other ones.
  - However, all the deadlocks are visited, and if there exists a state where an assertion is violated, the search will visit a state where the same assertion is violated
  - Limitation of this result to *acyclic* models… but successful experiments with cyclic models and bounded DFS
- Successful analysis of several software products developed in Lucent Technologies
  - Example: CDMA cell-site processing library, 100 KLOC

When is software model-checking effective?

Data-intensive systems↘	Digital signal processors Floating point units Graphical processors	Verifying compiler Financial software
Control-intensive systems↗	Cache coherence protocols Bus controllers	Embedded software Device drivers
	Hardware	Software

[Clarke et al. 2005]

? ? ?



Run-time verification, Model-checking programs, Coverage in model-checking, Bounded model-checking, Model-based testing,…

Zoom in on

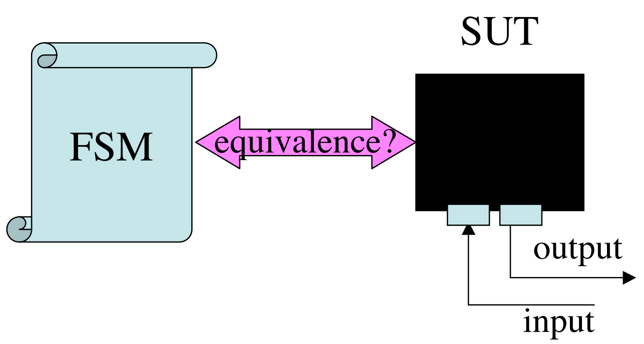
- What people call Software Model Checking
- What people call Model Based Testing
- What is Test Generation using Model-checking

Model-based testing

- Heavily overloaded term
  - There is a MBT workshop associated with ETAPS, but…
    - Almost everything is considered as a model (may be not wrong ☺)
- Models considered here:
  - Annotated graphs as in slide 4
  - Finite State Machines (FSM), possibly extended (EFSM)
  - Labelled Transition System (LTS), possibly with distinction between inputs and outputs
  - Back to [Chow 78] for FSM,
    - recommended reading : Lee and Yannakakis survey [1996]
  - and [Brinksma 88] for LTS, and then many others

Back in history: testability hypothesis

- System under test
  - unknown, but…



- Hypothesis:
- *the System Under Test behaves like some (unknown) FSM with the same number of states as the description*
  - In other words, in the SUT, whatever the execution path leading to some state s, the execution of transition *s –x:y-> s'* has the same effect (same output + same change of state)

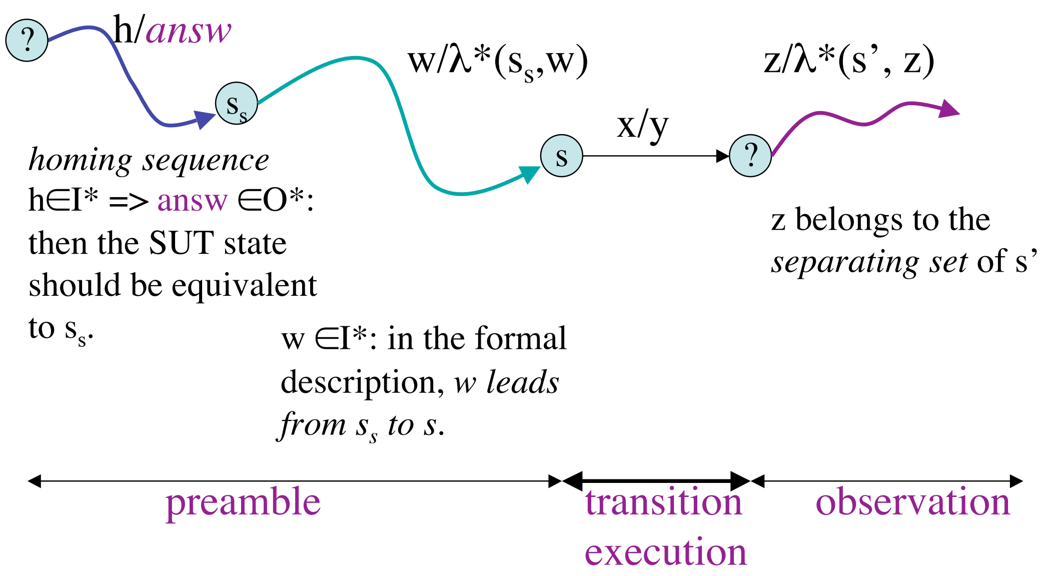
Back in history: control and observation

- A popular test strategy: transition coverage

  s –x:y-> s'   *is a transition. In state s, input x must produce output y and move to state s'*

- Questions
  - control: how to put the System Under Test into a state equivalent to *s*?
    - solution 1: reliable(??) reset, and then adequate input sequence
    - solution 2: "homing sequence", and then adequate input sequence
  - observation: how to check that after receiving *x* and issuing *y*, the SUT is in a state equivalent to *s'*?
    - « separating family » : collection $\{Z_i\}_{i=1,\dots,n}$ of sets of input sequences whose output sequences make it possible to distinguish s_i from any other state

One of the tests for s -x/y-> s'

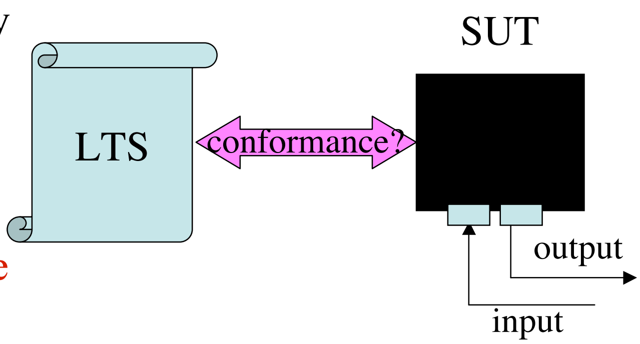


A strong result

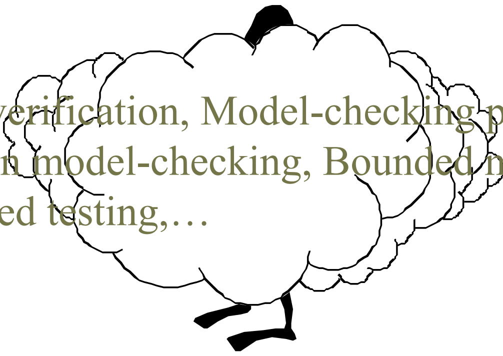
- Checking sequence:
  - covers every transition and its separating set; *distinguishes the description FSM from any other FSM with the same number of states*
- Finite, *but may be exponential…*
  - in length, construction
- Exhaustivity
  - transition coverage is ensured
- Control
  - homing sequence, or reliable reset
- Observation
  - distinguishing sets, or variants (plenty of them!)

The LTS approach

- Transitions are labelled by actions
- *Concurrent composition and synchronisations* are the key issues
- Conformance relations are no more equivalence, but various testing preorders
- Strong results on derivation of exhaustive test sets and selection
- On-line or off-line derivation methods



? ? ?



Run-time verification, Model-checking programs, Coverage in model-checking, Bounded model-checking, Model-based testing,…

Zoom in on

- What people call Software Model Checking
- What people call Model Based Testing
- What is Test Generation using Model-checking

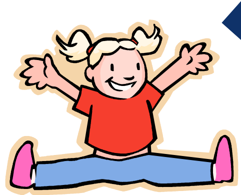
Test generation using model-checking

- Exploits the fact that model-checkers may yield counter-examples
  - Given φ, a required property of the SUT
  - Given a model M of the SUT
  - Model-check M for ¬ φ
- The model-checker will reject ¬ φ and produce a counter-example, i.e. a trace that satisfies φ, i.e. a test sequence for φ
  - Popular, most model-checkers have been experienced for test generation
  - Nice, but…

New issues, …and good old ones

- φ must be a formula in some temporal logic (not always convenient)
- An example:
  - φ: *AG(¬request ∨ (request U grant))*
  - ¬ φ: *EF(request ∧ ¬(request U grant))*
  - One counter-example is not enough (because of the universal quantification) => exhaustivity and coverage issues
- The finite model is an over-approximation of the system
  - Feasability, constraint solvers…

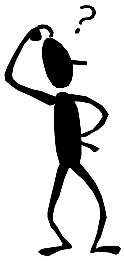
Conclusion (1)



- Significant advances in the three domains
  - Each one makes use of the other ones in some occasions
  - Very good specialised tools
- A politically correct and frequent comment:
  - All these methods are now used together, and this convergence will lead to great results
  - Model-checking is very powerful and solves most problems in static analysis and model based testing and more generally in verification

Conclusions (2)

- We are not so far…
- Many tricky scientific issues, among many others:
  - Standard temporal logics can specify only regular properties; correctness of procedures w. r. t. pre- and post conditions are not regular [Alur 2005]… *Integration of model-checking and program proving is not as clear as it is claimed by some authors.*
  - Constraint solving remains a bottle-neck: *most success stories on large programs static analysis or testing are either limited to linear arithmetic, or not fully automated*
  - Dealing with the "abstraction gap" in proving (reasoning with equality) and testing (oracle) is not solved in general



Final conclusion

It is not because a problem is undecidable that one must not attack it.

Be cautious about miracles