How do we know that our system is correct?

Hana Chockler
Department of Informatics
King's College, London

How to verify computerized systems?

Bug hunting

Certification of correctness

Verified

How to verify computerized systems?

Bug hunting

Certification of correctness

Simulation-based verification/ Testing

checking the behavior of a system by executing it many times

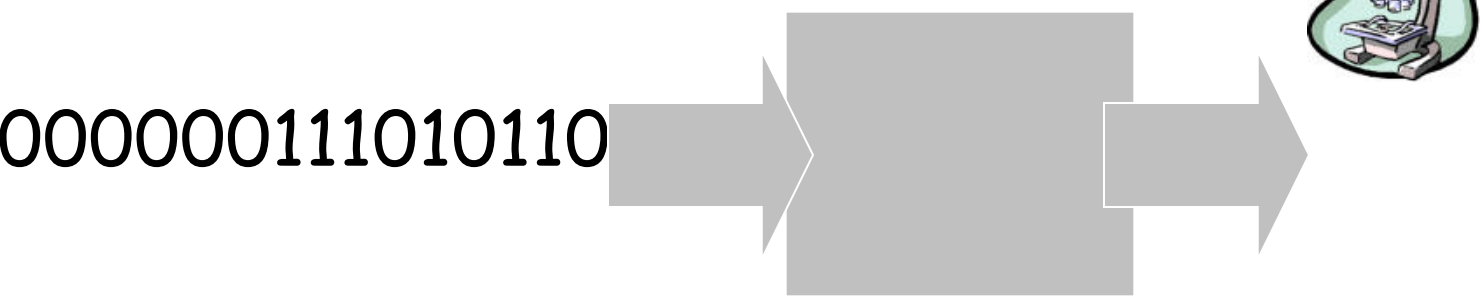
- This process is **sampling**
- The sampling should be as exhaustive as possible
- **Coverage metrics** are widely used as heuristic measures of exhaustiveness of sampling

Formal methods (Model checking)

formally and automatically proving the correctness of a system according to a predefined specification

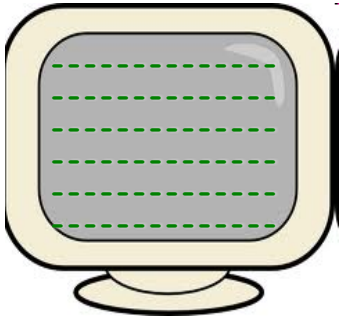
Simulation-based Verification

Execute the system in parallel with a reference model…



…with respect to some input sequences.

Testing



Execute the program on a test suite and inspect manually or (semi-)automatically

" To know that we know what we know, and that we do not know what we do not know, that is true knowledge ." Confucius

Simulation-based Verification and Testing

Observations:

- The simulation/testing process is **sampling**
- The sampling should be as exhaustive as possible
- The process can only find bugs – there is no way to guarantee that the system under test is correct

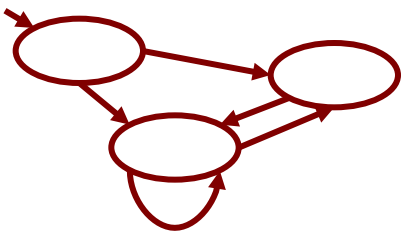
How exhaustive is the sampling?
Did we check all locations of
most complex functionality?

Various coverage metrics are
widely used as heuristic
measures of exhaustiveness
of verification

Formal Verification (Model Checking)

Is the system correct?

A mathematical model of the system M (an FSM)



verified

A formal specification φ

Does M satisfy ϕ?

no

counter example

yes

Do we really know this?

Claim: Verification is only as good as the specification

Best possible specifications generated automatically

learning

causality and responsibility

filtering redundant information

vacuity

coverage

query checking

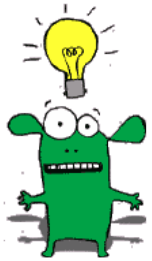
This is what I thought in 2015

Claim: Verification is only as good as the specification

Pass in Model Checking: Is it really correct?

Model checking ≠ sampling
The whole reachable state space is visited

Do we know that the system is correct if model checking passes?



Correctness of the "pass" result depends on **correctness** and **exhaustiveness** of the specification

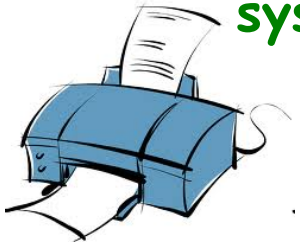
Did I check **what** I wanted to check?

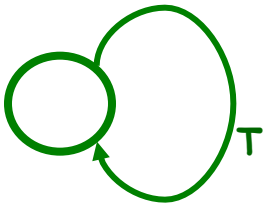
Did I check **everything** I wanted to check?

Suspecting a positive answer [IBM, Intel]

φ = always (req -> eventually grant)

system M:





Printer that doesn't print

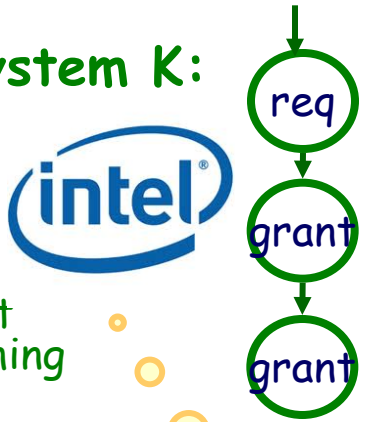
Does M satisfy φ?

IBM

Yes, but in a non-interesting way!

vacuity

system K:



intel

Printer that prints everything twice

Does K satisfy φ?

Yes, but maybe in an unexpected way!

coverage

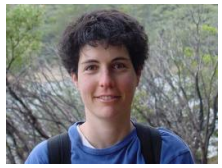
The story of vacuity

1997

Beer, Ben-David, Eisner, Rodeh
The first definition of vacuity

1999

Kupferman, Vardi
Formal definitions and complexity



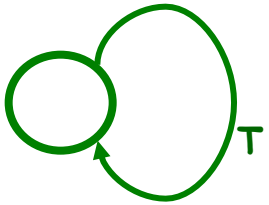
timeline

Vacuity – the main idea

Vacuous satisfaction of φ in M means that some part of φ is irrelevant in M

Printer that doesn't print

system M:



Means that φ can be strengthened without falsifying it in M

φ = always (req -> eventually grant) ➡ φ ' = always (req -> eventually false)

strengthening

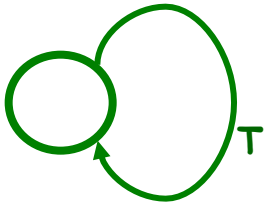
Another way to see it: can we introduce mutations (changes) to φ without falsifying it in M?

Vacuity – the main idea

Vacuous satisfaction of φ in M means that some part of φ is irrelevant in M

Printer that doesn't print

system M:



Means that φ can be strengthened without falsifying it in M

φ = always (req -> eventually grant)

mutation

mutant φ '

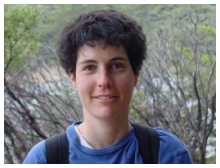
Non-affecting mutations can point to problems

Number of mutations depends on the size of φ and the types of mutations

The story of vacuity

1997 Beer, Ben-David, Eisner, Rodeh
The first definition of vacuity

1999 Kupferman, Vardi
Formal definitions and complexity



Lots and lots of papers, studying vacuity for different logics, different languages, with different definitions …

2007 Ben-David, Fisman, Ruah:
Vacuity for regular expressions

2008
Chockler, Gurfinkel, Strichman
The Strongest Passing Formula

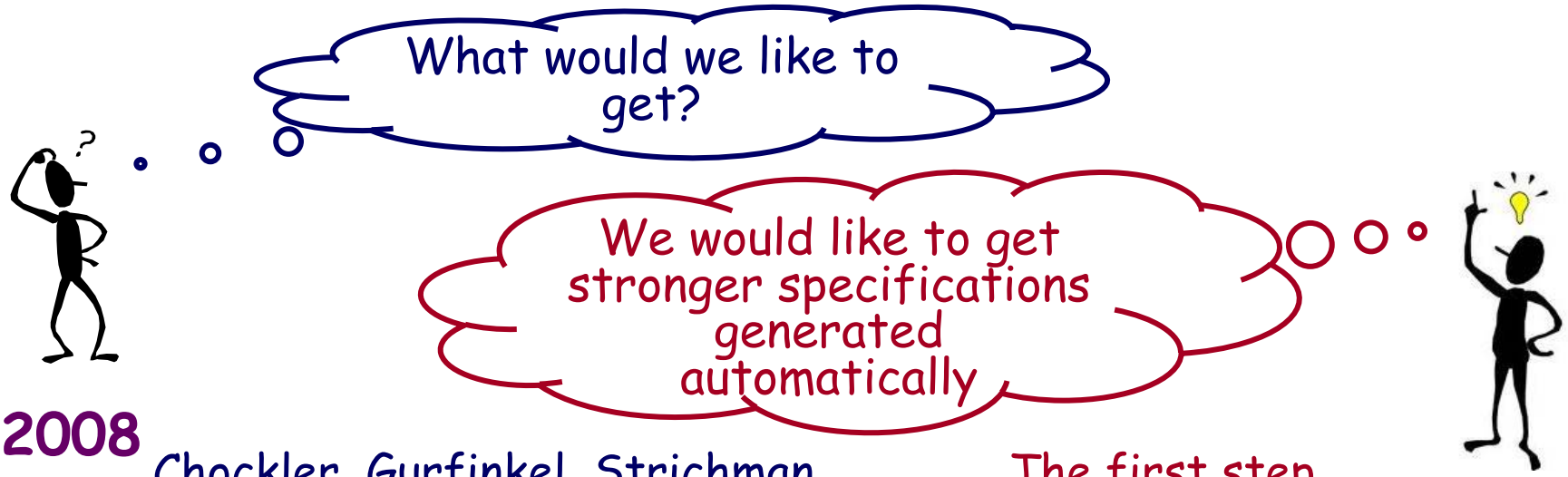
Vacuity is a part of all commercial model-checking tools
(IBM, Intel, Cadence and others)

What is the output of vacuity check?

Vacuous satisfaction of φ in M means
that some part of φ is irrelevant in M

Standard vacuity check outputs the parts of φ
that can be replaced by ?

φ = always (a OR b OR c) → vacuity → a is redundant:
φ' = always (b OR c) holds



2008

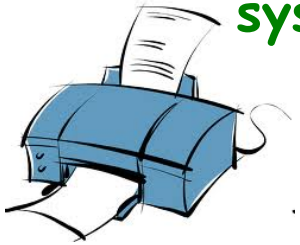
Chockler, Gurfinkel, Strichman
The Strongest Passing Formula

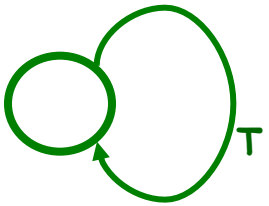
← The first step in this direction

Suspecting a positive answer [IBM, Intel]

φ = always (req -> eventually grant)

system M:





Printer that doesn't print

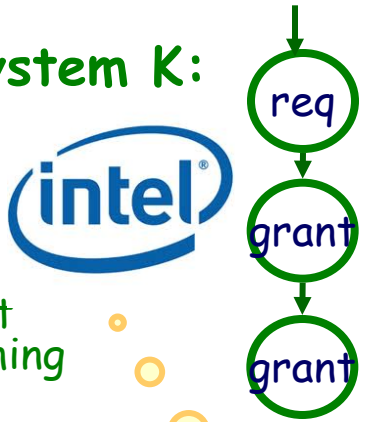
Does M satisfy φ?

IBM

Yes, but in a non-interesting way!

vacuity

system K:



intel

Printer that prints everything twice

Does K satisfy φ?

Yes, but maybe in an unexpected way!

coverage

The story of coverage

1999

Hoskote, Ho, Kam, Zhao
The first definition of coverage for model checking

2001

Chockler, Kupferman
Formal definitions and algorithms

timeline

Coverage

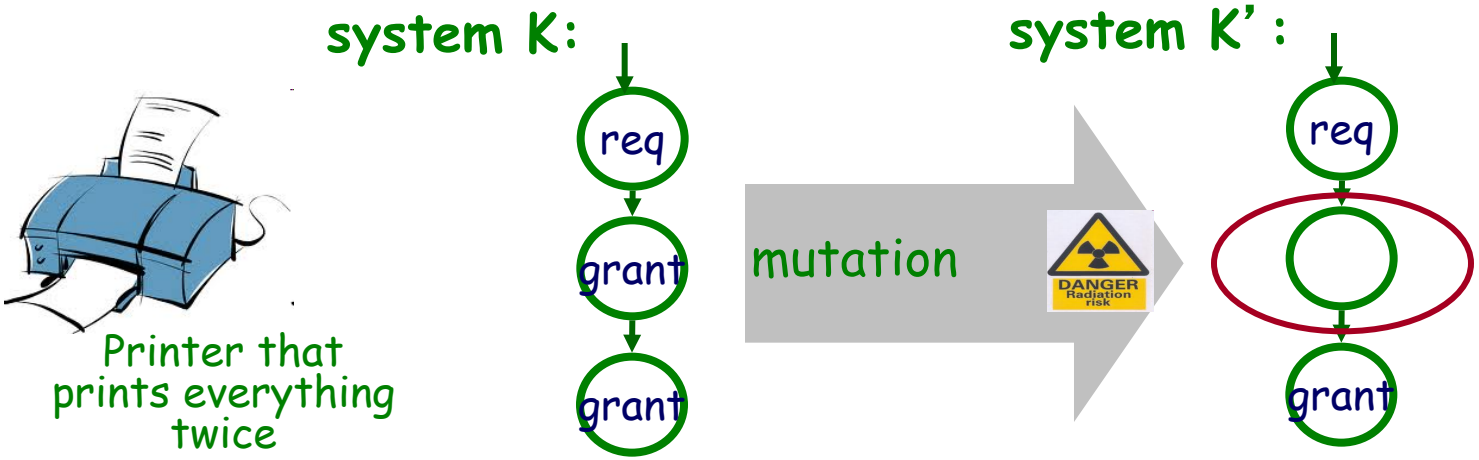
The most general definition:

In model checking: an element is covered
↔
The system with the element modified (***mutated***) no longer satisfies the specification

- elements are small (atomic) - mutations are **small changes**
- if a mutant system still satisfies the specification ➔ we did not check this particular corner of the system

Coverage

φ = always (req -> eventually grant)



A non-covered mutation represents a "corner" of the system that was not verified
(and hence may contain bugs)

The story of coverage

1999

Hoskote, Ho, Kam, Zhao
The first definition of coverage for model checking

2001

Chockler, Kupferman
Formal definitions and complexity

Some papers proposing new definitions and algorithms

timeline

Different types of mutations

- Flipping the value of one output signal in one state (Intel)
- Same with control signals
- Freeing a signal (turning it to an input)
- VHDL code mutations (erasing or changing a line)
- Removing behaviors (by removing states or otherwise)
- Changing one net in the net-list

mutations are small changes – either in the signals in the representation inside the tool, or in the code written by the designer

The story of coverage

1999

Hoskote, Ho, Kam, Zhao
The first definition of coverage for model checking

2001

Chockler, Kupferman
Formal definitions and complexity

Some papers proposing new definitions and algorithms

Complexity of computing coverage

Manual effort in examining the results

timeline

Problem: how to compute coverage efficiently?

Model-checking each mutant system separately is infeasible

Useful observation – all mutant systems are similar!

This leads to feasible and efficient algorithms:

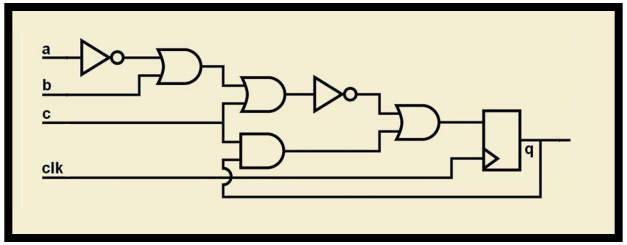
- **Symbolic algorithms:** represent all possible mutants together by adding a small number of symbolic variables
- **Improving average complexity**: consider possible mutations as unknown values and attempt to compute as much as possible without assigning them

Was implemented and actually works at IBM

- **Reusing** the results of verification of the original system:
  - **Interpolation-based** coverage computation: save the proof and reuse it
  - **ic3-based** coverage computation: save the high-level proof or symbolic partial counterexample and re-use it

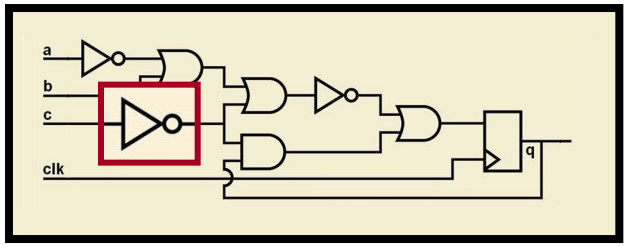
Reusing the results of verification

- Main idea: the same proof works for most mutant systems
- If the proof does not work, it can be "patched" to work – cheaper than re-verification



DUV

mutation



Mutant DUV

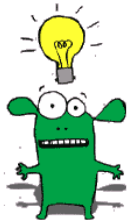
Proof

---

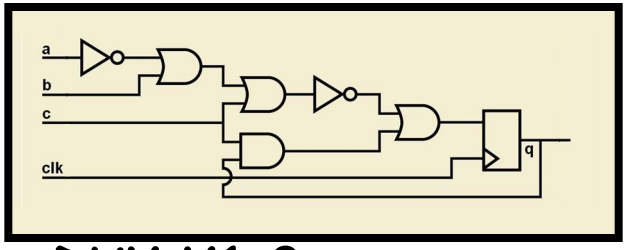
mutation

Patched Proof

A byproduct – efficient regression verification methodology

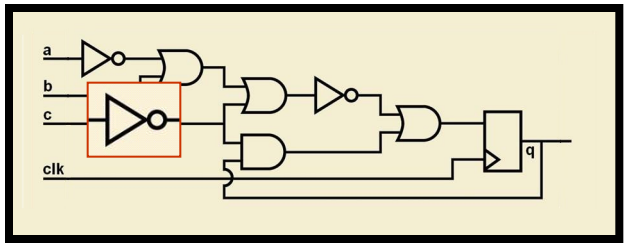


- Proofs and counterexamples are general enough to work for many mutant systems – in particular, for a new version
- If they don't work, they can be "patched" to work – cheaper than re-verification

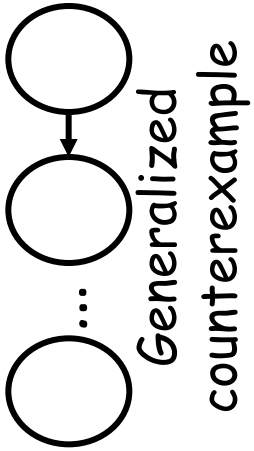


DUV V1.0

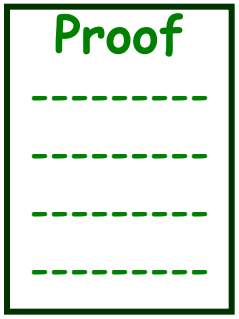
New version



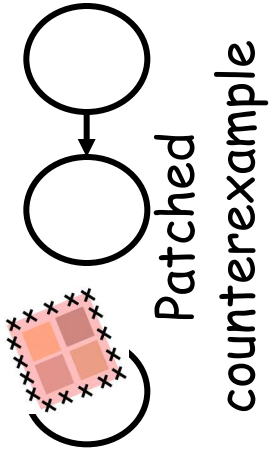
DUV V2.0



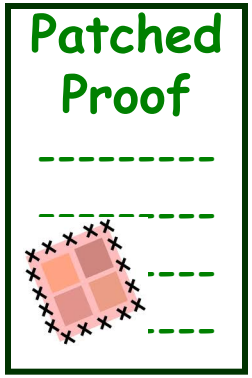
Generalized counterexample



small patch



Patched counterexample



The story of coverage

1999

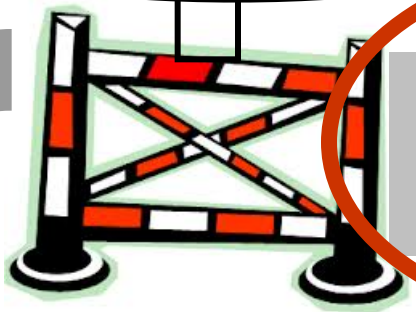
Hoskote, Ho, Kam, Zhao
The first definition of coverage for model checking

2001

Chockler, Kupferman
Formal definitions and complexity

Some papers proposing new definitions and algorithms

Complexity of computing coverage – Solved!



Manual effort in examining the results

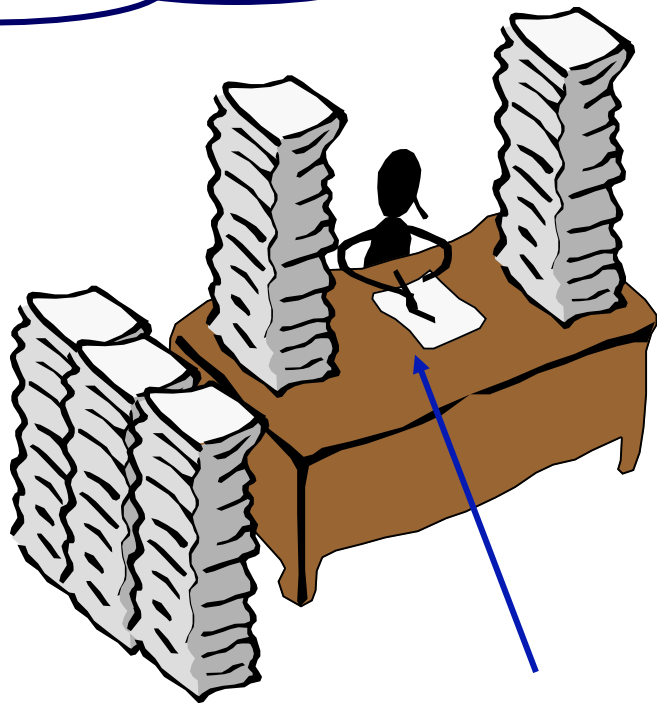
Results of the feasibility check – done at IBM

Mutations of a hardware design

There exist many non-covered mutations

Even in a thoroughly verified design!

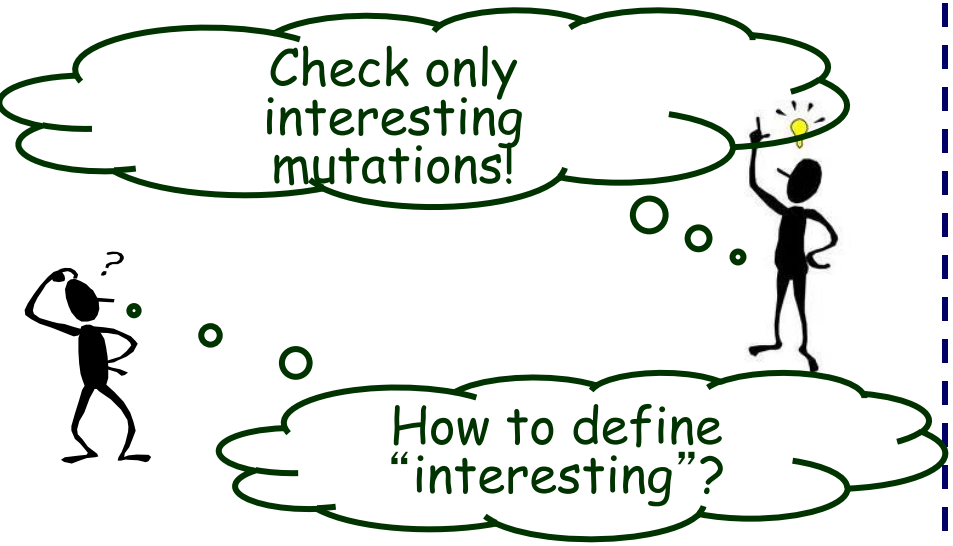
- There are hundreds of thousands of automatically generated mutations
- Not all of them are interesting
- Those that are interesting should be examined by the verification engineer and/or the designer
- Not all non-covered mutations point to a bug – but some do



unhappy designer

How to minimize the manual effort in checking coverage results?

Reduce the number of mutations



- Based on the structure of the design or on designer's intent
- Does it save power?
- Is it likely?
- Refinement of coverage

Automatic analysis of results

- Identify non-covered areas of the design
- Construct non-covered traces
- Suggest properties that have better coverage

The holy grail of sanity checks:

The story of coverage

1999

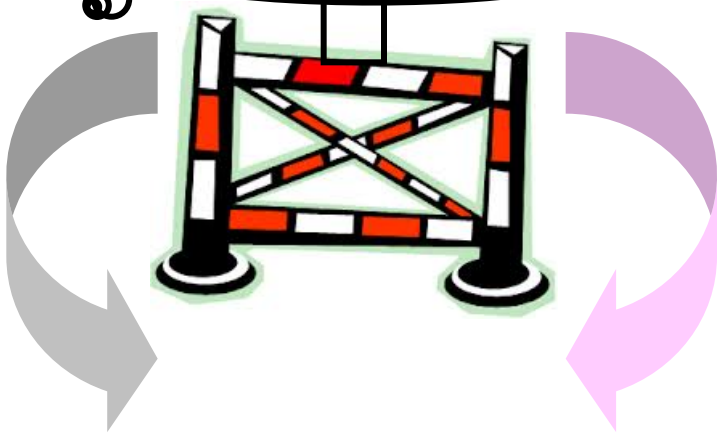
Hoskote, Ho, Kam, Zhao
The first definition of coverage for model checking

2001

Chockler, Kupferman
Formal definitions and complexity

Some papers proposing new definitions and algorithms

Complexity of computing coverage – Solved!

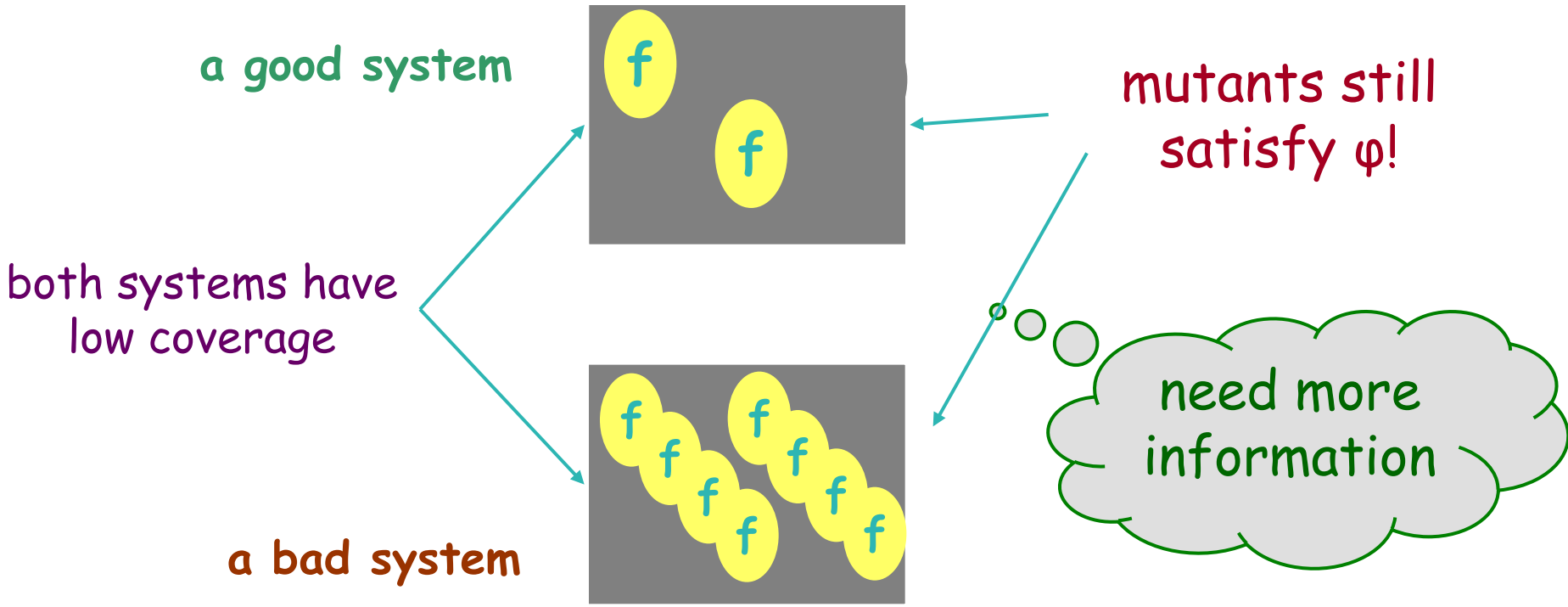


Manual effort in examining the results – First steps

Do we need a metric that refines coverage?

Problem: some specifications almost **always** have low coverage!

Example: φ = "f is computed at least twice"
(for fault tolerance)

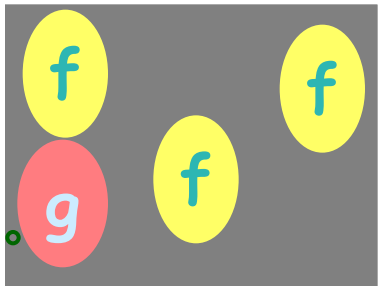


Replacing coverage with causality and responsibility (from AI):

And element is covered if its responsibility is 1.

Example: φ = "**f** is computed at least twice" (for fault tolerance)

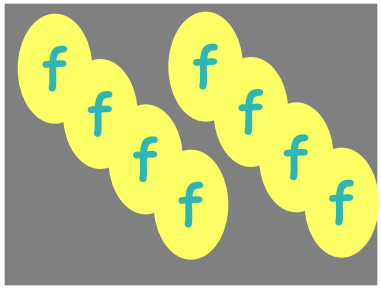
a good system



responsibility of each **f** for **ϕ** is **1/2**

responsibility of **g** for **ϕ** is **0**

a bad system



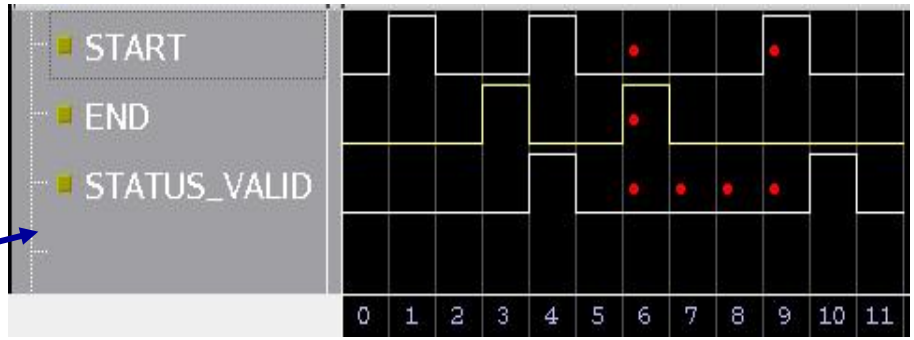
responsibility of each **f** for **ϕ** is **1/7**

for formal verification and other things ✓

Side note:
causality and responsibility are useful in general

Explaining counterexamples using causality (putting red dots):

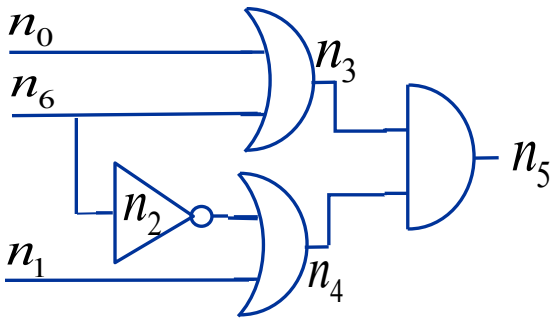
Part of the IBM formal verification product



φ = G((¬START ∧ ¬STATUS VALID ∧ END) → X[¬START U (STATUS VALID ∧ READY)]).

Symbolic trajectory evaluation (STE) refinement:

Implemented in Intel

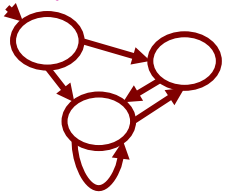


Best possible systems synthesized automatically

requirements

Double work!

A mathematical model of the system M



A formal specification φ

Does M satisfy ϕ?

the system is correct!

no

yes

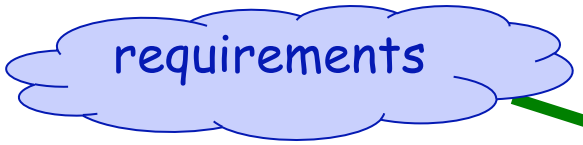
counter example

What if we can just generate a correct system automatically from the specification?

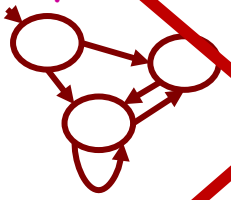
Synthesis

What if we can just generate a correct system automatically from the specification?

Synthesis



A mathematical model of the system M



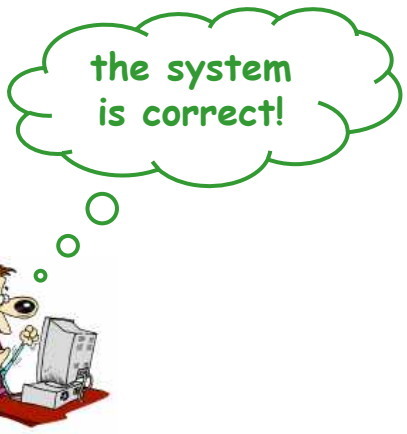
A formal specification φ

Does M satisfy ϕ?

no

counter example

yes



What if we can just generate a correct system automatically from the specification?

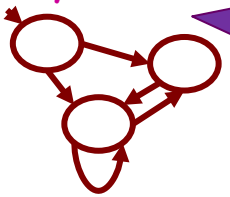
Synthesis

requirements

Best possible systems synthesized automatically

So the problem is solved? Can we all go home now?

A mathematical model
the system M



specification
φ

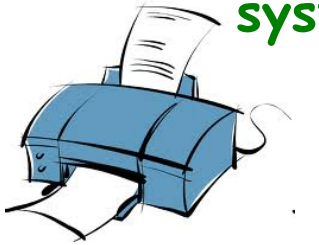
the system is correct by construction!

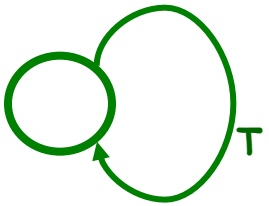
Not so fast…

Suspecting a positive answer [IBM, Intel]

φ = always (req -> eventually grant)

system M:





Printer that doesn't print

Does M satisfy φ?

IBM

Yes, but in a non-interesting way!

vacuity

system K:

Printer that prints everything twice

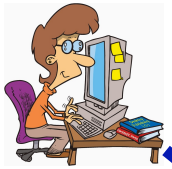
Does K satisfy φ?

Yes, but maybe in an unexpected way!

coverage

What happens when the specification is incomplete?

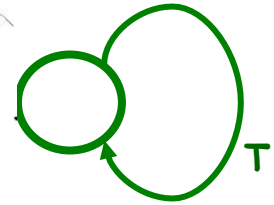
We need to print some stuff



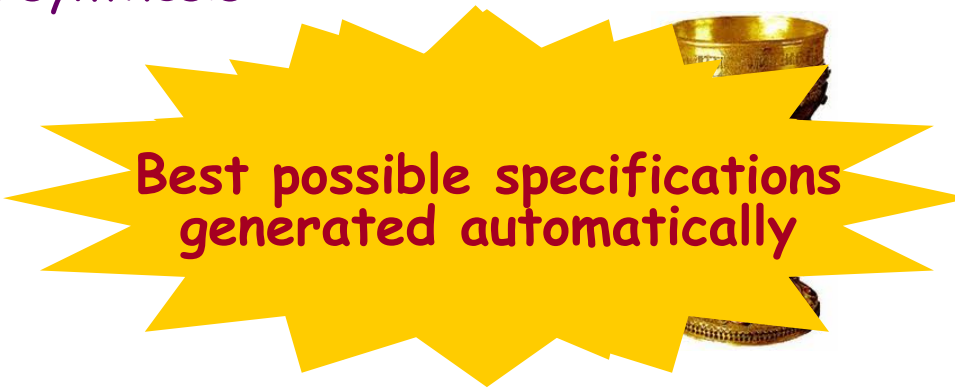
φ = always (req-> eventually grant)

Automatic synthesis

Printer that doesn't print (not connected)



vacuity



Synthesizing non-vacuous systems [BCES17]

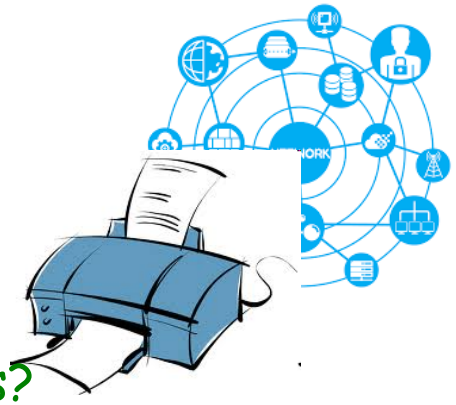
Vacuity is a non-interesting pass
of a specification in the system.
But we don't have a system yet! What do we do?

- Given a specification φ, strengthen it so that it does not allow vacuous satisfaction.
  - "There must exist an interesting behaviour".
- This results in a formula with existential quantifiers.
- Synthesize the resulting formula.
- Improve the system iteratively until there are many interesting behaviours.

Synthesizing non-vacuous systems [BCES17]

φ = always (req-> eventually grant)

Must allow any behaviour of inputs (req)
(that is, the printer is connected)

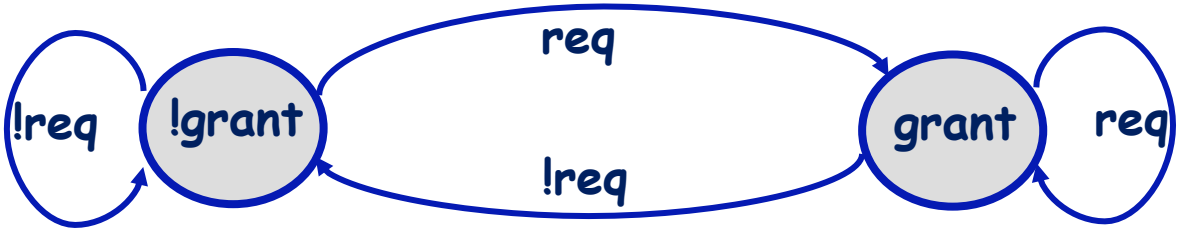


What about a printer that prints regardless of requests?

ψ = always (eventually(grant))

Must be at least one execution where eventually it stops printing

$\neg\psi$ = $\exists$ (eventually (always(!grant)))



Print ↔ a printing request arrives

So can we switch to synthesis and stop designing systems?

Not yet…

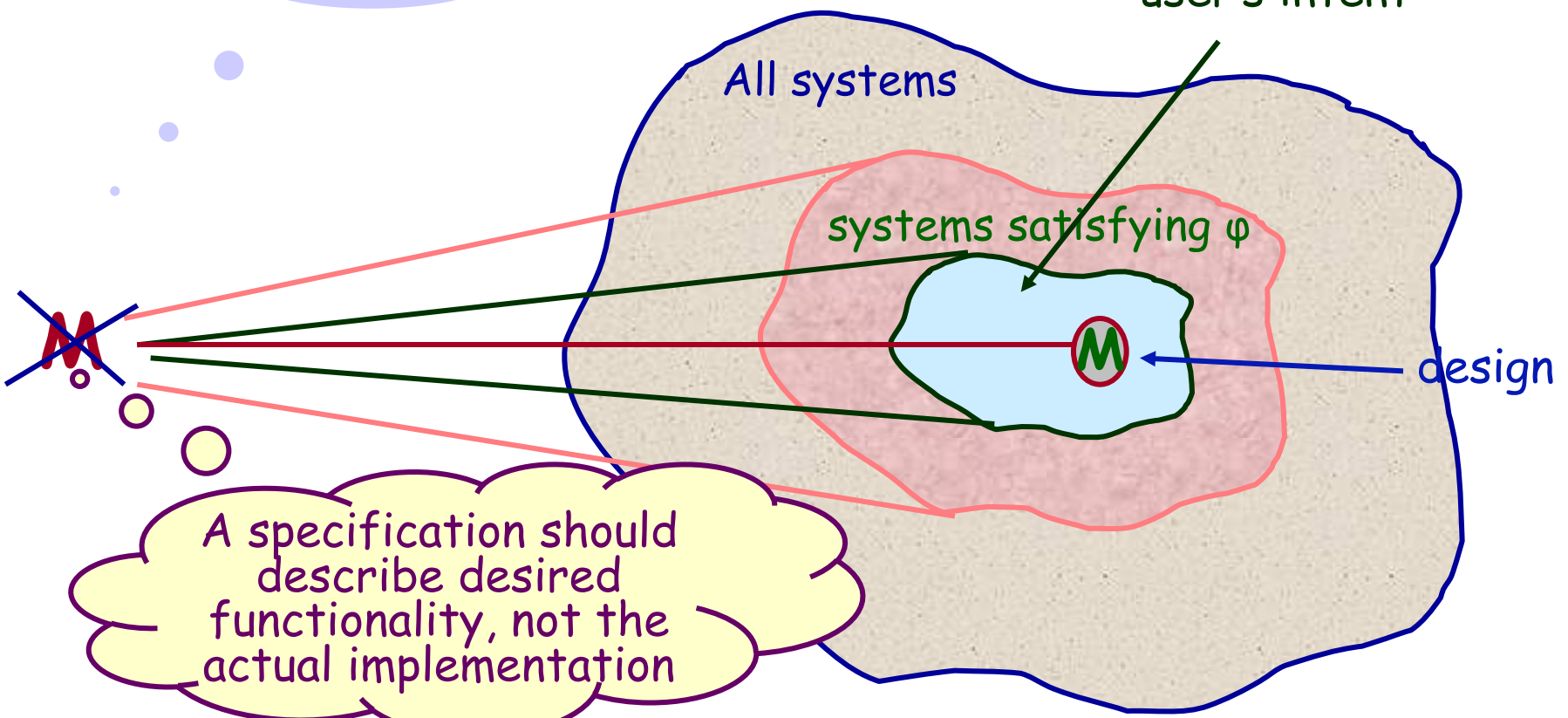
- Synthesis is double-exponential in the size of the specification
- We only know how to do non-vacuous bounded synthesis
- Nobody knows how to synthesize software
- **We don't know how to generate the best possible specifications automatically**

But hopefully in the future

The main goal of sanity checks:

Best possible specifications generated automatically

Strengthening and tightening




Questions?