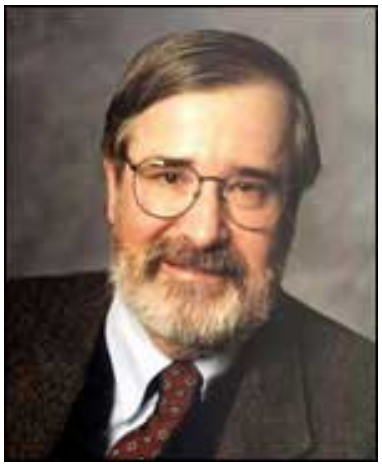
Formal Methods Explained

Kristin Yvonne Rozier
Iowa State University

Applied Formal Methods
August 22, 2021

Who Are They?

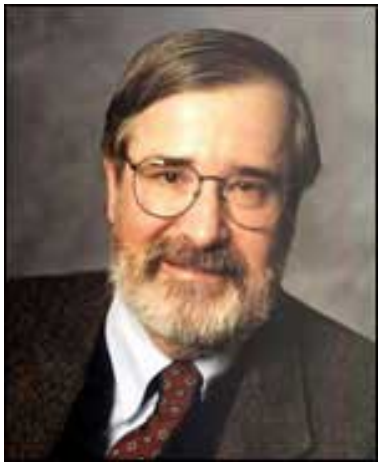


Edmund M. Clarke

E. Allen Emerson

Joseph Sifakis

The 2007 Turing Award

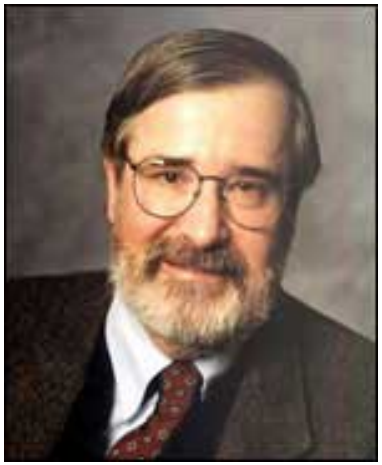


Edmund M. Clarke

E. Allen Emerson

Joseph Sifakis

The 2007 Turing Award



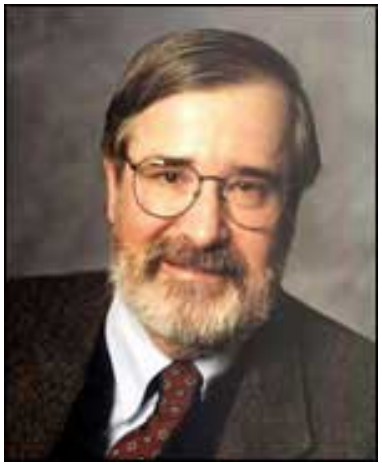
Edmund M. Clarke

E. Allen Emerson

Joseph Sifakis

For developing Model-Checking into a highly effective verification technology, widely adopted in the hardware and software industries.

The 2007 Turing Award



Edmund M. Clarke

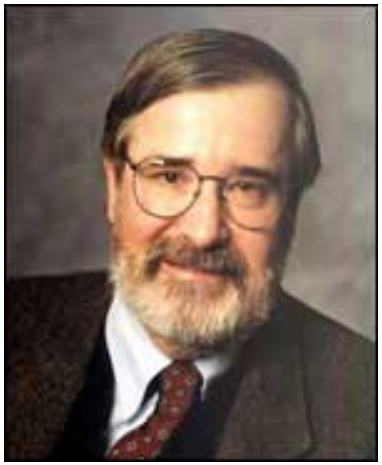
E. Allen Emerson

Joseph Sifakis

For developing Model-Checking into a highly effective verification technology, widely adopted in the hardware and software industries.

- What are Model Checking, Formal Methods in general?

The 2007 Turing Award



Edmund M. Clarke

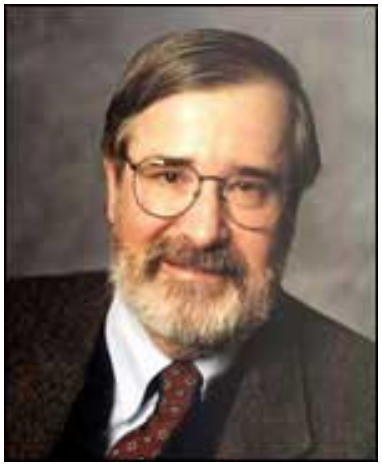
E. Allen Emerson

Joseph Sifakis

For developing Model-Checking into a highly effective verification technology, widely adopted in the hardware and software industries.

- What are Model Checking, Formal Methods in general?
- Why do we need Formal Methods?

The 2007 Turing Award



Edmund M. Clarke

E. Allen Emerson

Joseph Sifakis

For developing Model-Checking into a highly effective verification technology, widely adopted in the hardware and software industries.

- What are Model Checking, Formal Methods in general?
- Why do we need Formal Methods?
- Why don't we formally verify all systems?

Formal Methods

"Formal Methods" are mathematically rigorous techniques for the specification, design, and verification of software and hardware systems.

Formal Methods:

- Check that behaviors (statements in mathematical logic) always hold in a system (specified in a logical language)
- Symbolically examine the entire state space (all possible inputs)
- Provide absolute assurance of a correctness or safety property

Given formal definitions of what a system does (M) and what it should do (φ), formal methods can be used to show that M *satisfies* φ.

Formal Methods

"Formal Methods" are mathematically rigorous techniques for the specification, design, and verification of systems.

Formal Specification:

- System requirements (properties) in mathematical logic

Formal Verification:

- Check that requirements always hold in a system (specified in a logical language)
- Examine the entire state space (all possible inputs)
- Provide absolute assurance of a correctness or safety property

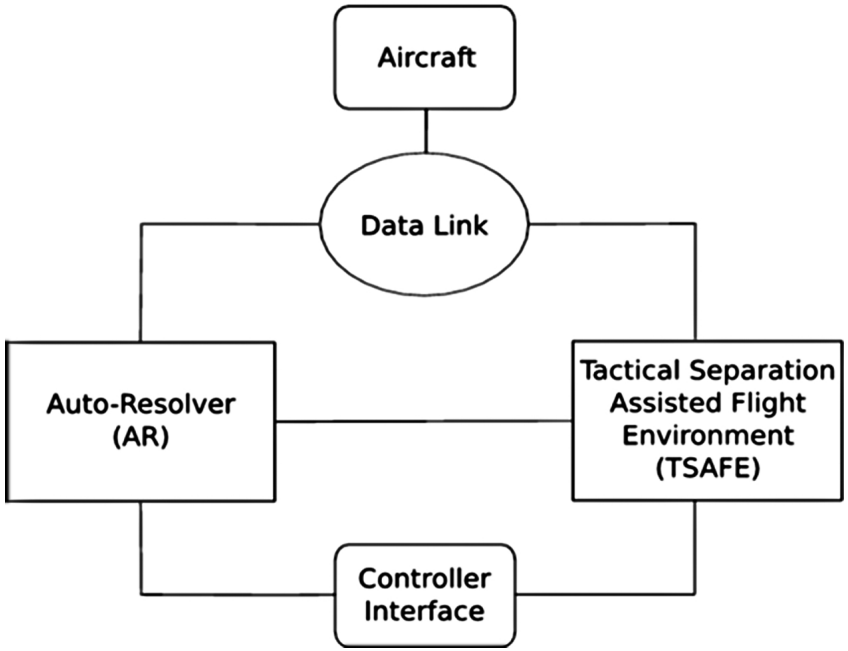
Given formal definitions of what a system does (M) and what it should do (φ), formal methods can be used to show that M *satisfies* φ.

Intuitively, the system does what you think it should do and nothing else.

Why Use Formal Methods?

Formal Verification of an NGATS Communication Protocol

M: What the system does



- The auto-resolver sends flight commands to the aircraft.
- The aircraft may request a specific flight command.
- The controller may request a specific flight command
- TSAFE commands override all others.
- Requests cannot be made in the presence of conflict.
- Only one request is registered at a time.

NGATS Communications Protocol Specifications

φ: How the system should behave

- ✓ *"Every conflict is addressed"*

  Or, we can choose to specify a stricter version of this property:
- ✓ *"Every conflict is addressed immediately (i.e in one time step)"*
- ✓ *"The system will never issue conflicting commands"*
- ✓ *"All conflicts are eventually resolved"*
- ✓ *"All controller requests are eventually addressed"*
- ✓ *"All aircraft requests are eventually addressed"*

We can verify all of these behaviors hold via *formal specifications . . .*

Propositional Logic Behavior Properties

Propositional Logic:

p, q	Boolean variables
$\neg p$	not
$p \wedge q$	and
$p \vee q$	or
$p \rightarrow q$	implies

Propositional Logic Behavior Properties

Propositional Logic:

p, q	Boolean variables
$\neg p$	not
$p \wedge q$	and
$p \vee q$	or
$p \rightarrow q$	implies

Continuous systems necessarily involve a notion of time. Propositional logic is not expressive enough to describe real systems.

Temporal Logic Behavior Properties

Linear Temporal Logic (LTL) formulas reason about linear timelines:

- a finite set *Prop* of atomic propositions
- Boolean connectives: $\neg$, $\wedge$, $\vee$, and $\rightarrow$
- temporal connectives:

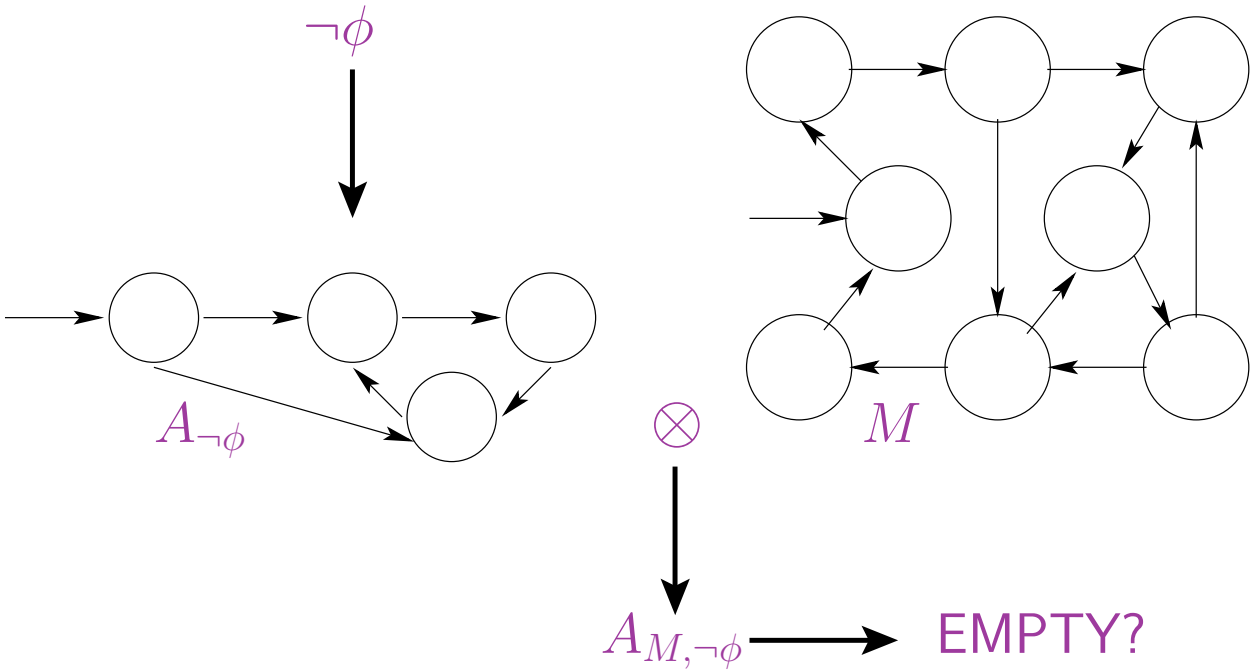
$\mathcal{X}\varphi$	NEXT TIME
$\varphi\mathcal{U}\psi$	UNTIL
$\varphi\mathcal{R}\psi$	RELEASE
$\Box\varphi$	ALWAYS, also called $\mathcal{G}$ for "globally"
$\Diamond\varphi$	EVENTUALLY, also called $\mathcal{F}$ for "in the future"

Computational Tree Logic (CTL) reasons about branching paths:

- Temporal connectives are proceeded by path quantifiers:

A	for all paths
E	exists a path

Model Checking



Model checking finds disagreements between the system model and the formal specification.

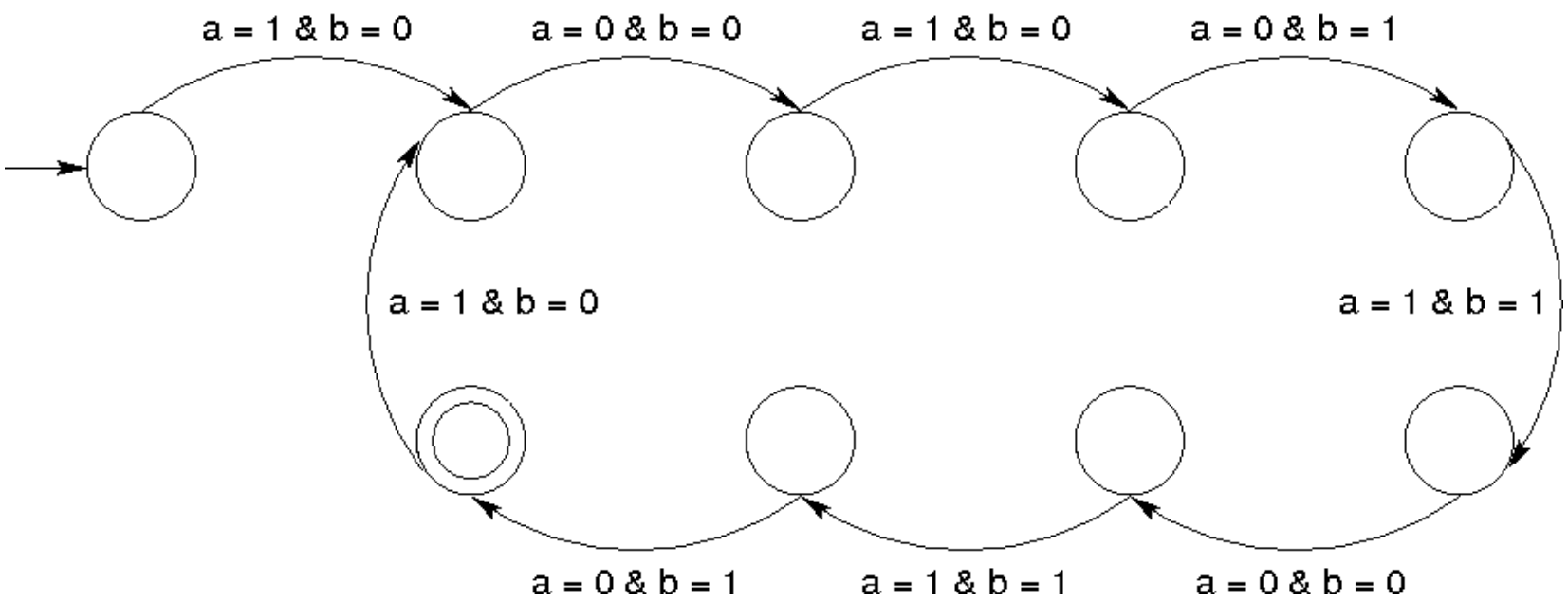
If there is disagreement, a *counterexample trace* is returned. Otherwise, the system satisfies the specification.

A Simple System Represented As An Automaton

2-bit Binary Counter:

a marks the the start of a new number

b is the bit counter

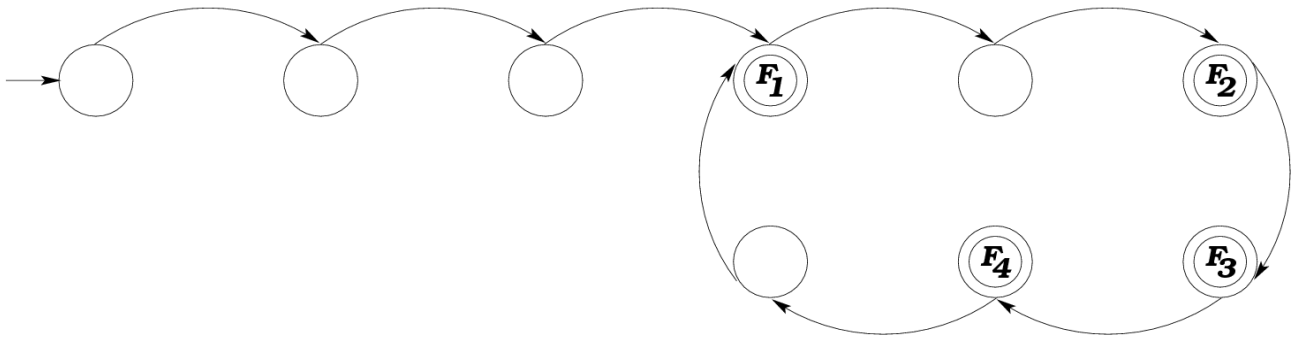


00 01 10 11 . . .

How Is Model Checking Implemented?

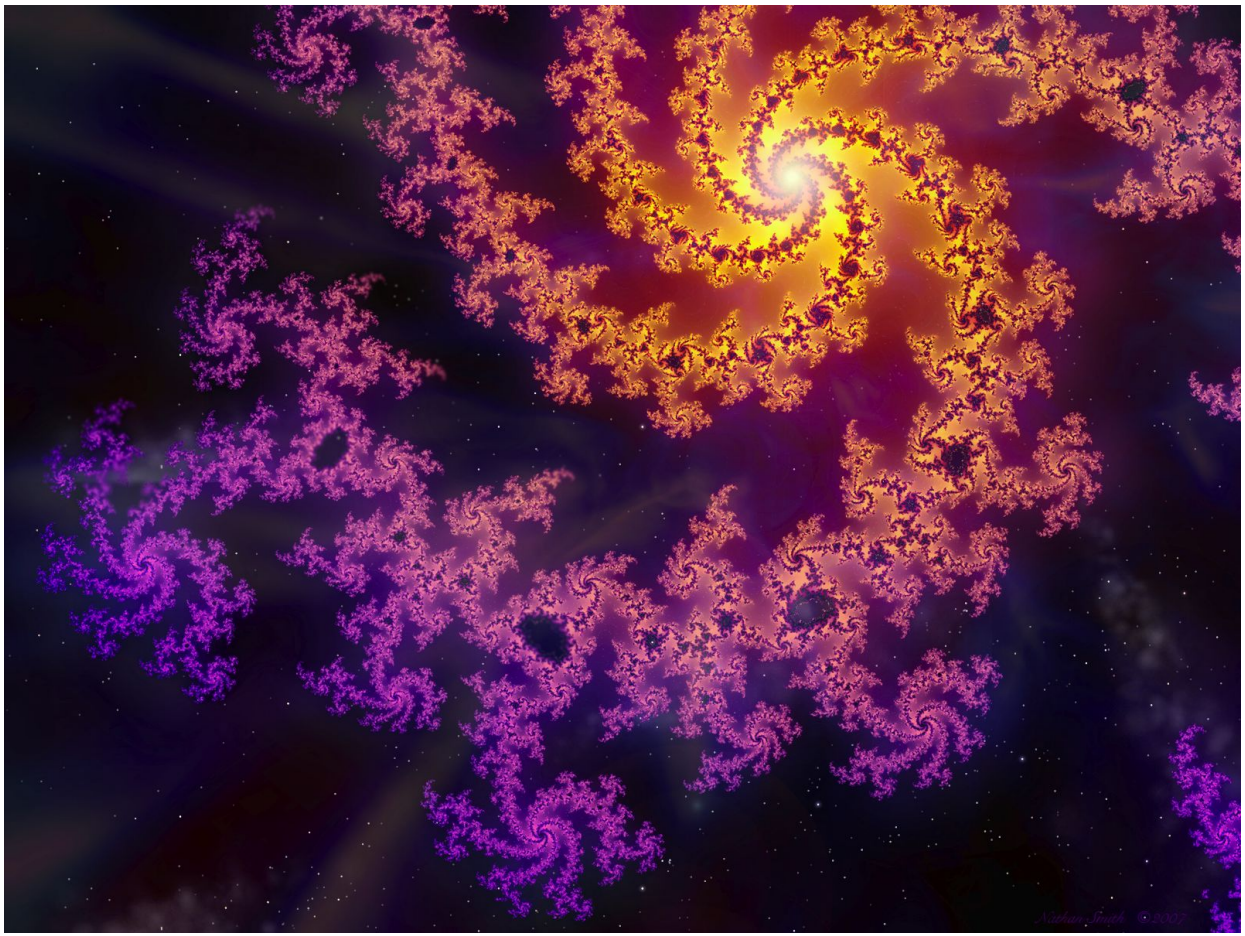
Explicit Model Checkers:

- Construct the state-space explicitly (i.e., create an automaton).
- Search for a trace falsifying the specification.
  - For finite (safety) properties, look for an accepting run
  - For nonterminating linear properties, look for an accepting lasso by finding strongly connected components in the automaton graph.



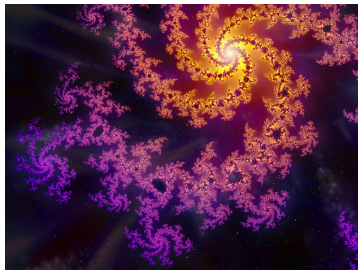
accepting lasso = counterexample trace

Explicit-state Model Checking: It's Like a Fractal . . .



Symbolic Model Checking: It's Like a Fractal Equation!

- Equations that capture all relevant aspects of the system design but reduce the state space.



$= \forall n, x_n = Frac(2^n x + 0)$

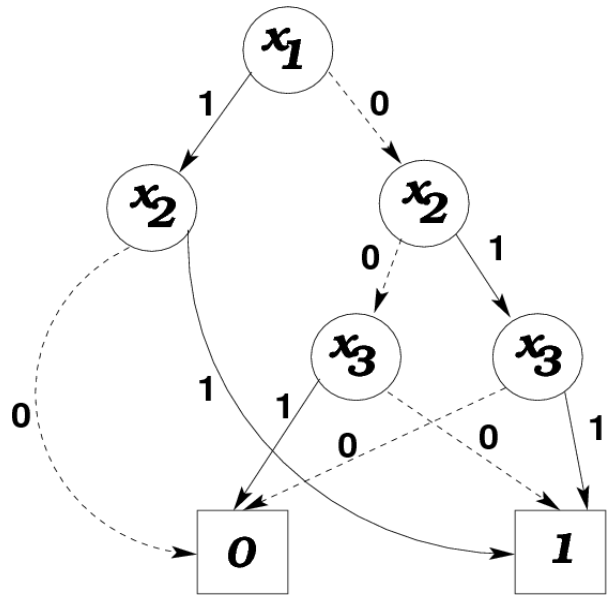
How Is Model Checking Implemented?

Symbolic Model Checkers:

- Represent the model symbolically using Boolean formulas.
- Analyze the model using Binary Decision Diagrams (BDDs) or Satisfiability (SAT).

Reasoning:

- increase efficiency
- decrease memory usage
- increase scalability
- increase speed



binary decision diagram

All software (including continuous systems) is executed over Boolean logic.

Theorem Proving

1. Describe the system in a formal language.
2. Satisfy type-checks and other proof obligations. (type-theoretic languages)
3. Introduce behavior properties as theorems that must be proven to hold using:
   - the formal description of the system behavior
   - a set of logical axioms
   - a set of inference rules

Use rigorous logical deductions (i.e. each step follows from a rule of inference and hence can be checked by a mechanical process.)

If the safety property does not hold, the programmer will encounter a proof step that cannot be discharged and which describes the circumstances of the bug.

Note this is a one-way implication: such a proof step could signify user error!

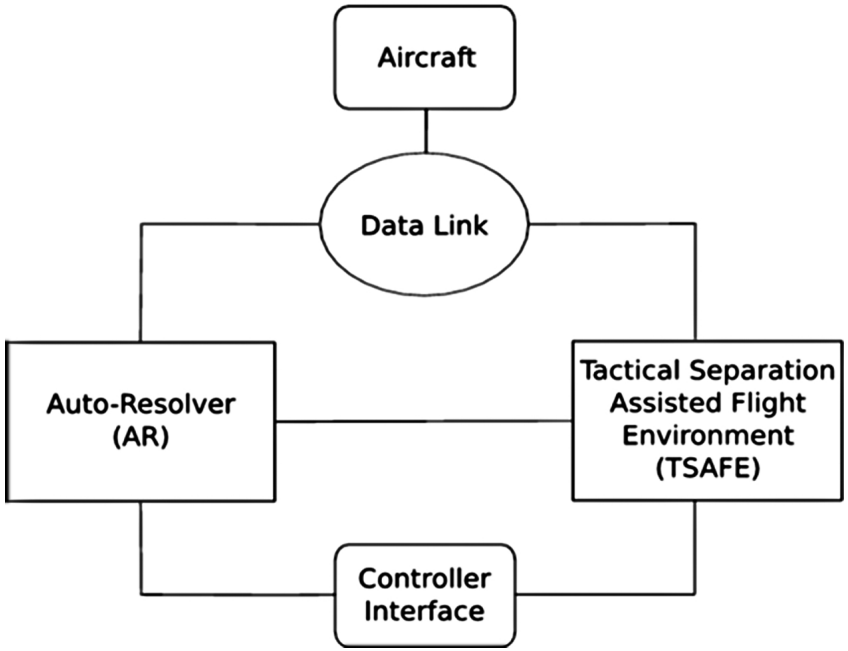
Example Theorems in PVS

Let p_1, p_2 be planes.
Let a be an action that a plane p can take.

Soundness (safety) for separation assurance:
$\forall p_1, p_2 : (good\_actions\_taken) \implies dist(p_1, p_2) > D$, where D is the minimum separation distance

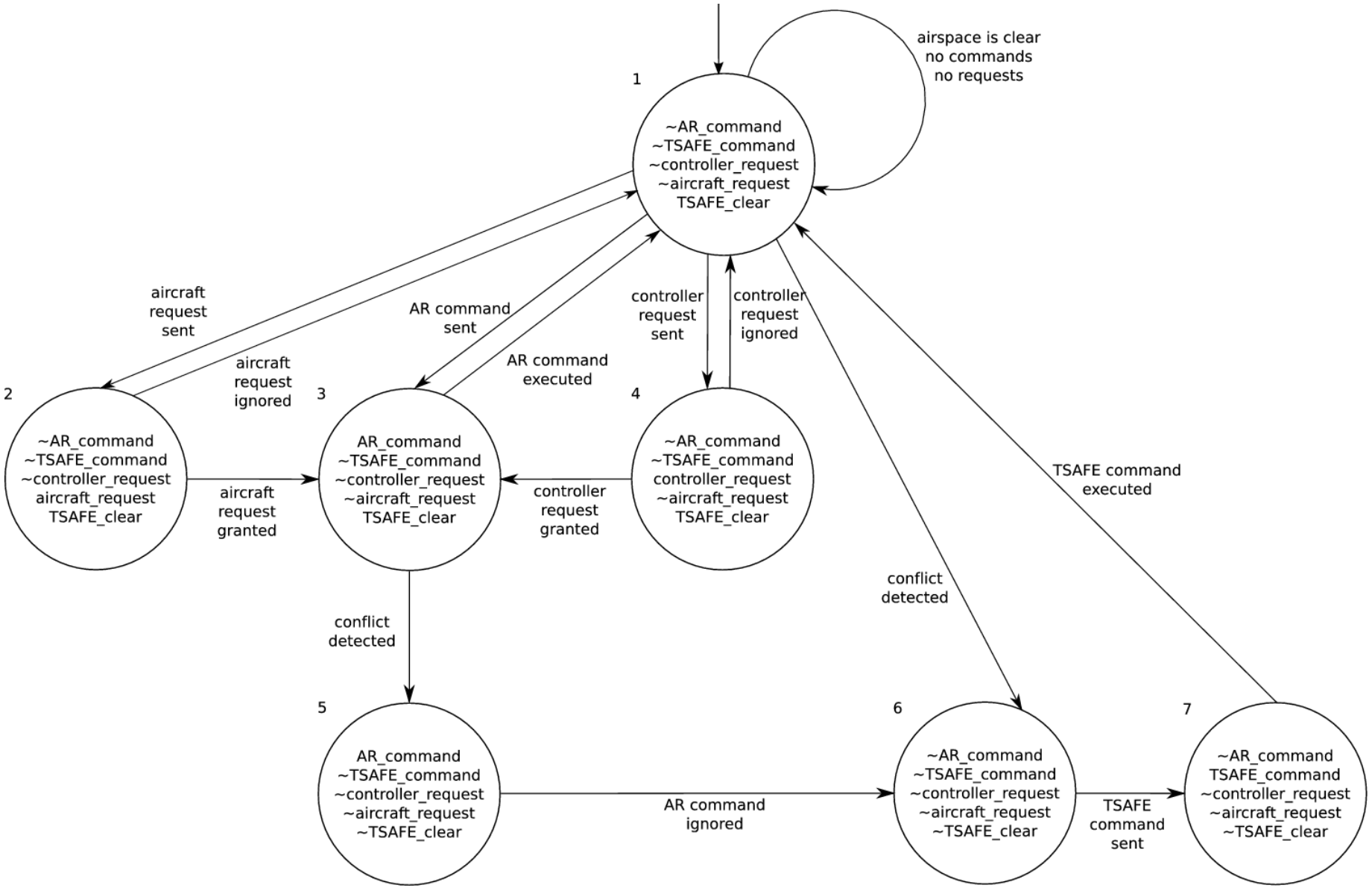
Completeness (liveness):
$\forall p \exists a : good\_action(a, p)$

Example: Verifying NGATS Communication Protocol



Variables
AR_command
TSAFE_command
controller_request
aircraft_request
TSAFE_clear

- The auto-resolver sends flight commands to the aircraft.
- The aircraft may request a specific flight command.
- The controller may request a specific flight command
- TSAFE commands override all others.
- Requests cannot be made in the presence of conflict.
- Only one request is registered at a time.


1
~AR_command
~TSAFE_command
~controller_request
~aircraft_request
TSAFE_clear
airspace is clear
no commands
no requests
2
~AR_command
~TSAFE_command
~controller_request
aircraft_request
TSAFE_clear
3
AR_command
~TSAFE_command
~controller_request
~aircraft_request
TSAFE_clear
4
~AR_command
~TSAFE_command
controller_request
~aircraft_request
TSAFE_clear
5
AR_command
~TSAFE_command
~controller_request
~aircraft_request
~TSAFE_clear
6
~AR_command
~TSAFE_command
~controller_request
~aircraft_request
~TSAFE_clear
7
~AR_command
TSAFE_command
~controller_request
~aircraft_request
~TSAFE_clear
aircraft request sent
aircraft request ignored
AR command sent
AR command executed
controller request sent
controller request ignored
aircraft request granted
controller request granted
conflict detected
conflict detected
TSAFE command executed
AR command ignored
TSAFE command sent

NGATS Communications Protocol Specifications

- *"Every conflict is addressed"*
  $\text{ALWAYS}(\neg \mathit{TSAFE\_clear} \rightarrow \text{EVENTUALLY}\ \ \mathit{TSAFE\_command})$
  Or, we can choose to specify a stricter version of this property:
- *"Every conflict is addressed immediately (i.e in one time step)"*
  $\text{ALWAYS}(\neg \mathit{TSAFE\_clear} \rightarrow \text{NEXT}(\mathit{TSAFE\_command}))$
- *"The system will never issue conflicting commands"*
  $\text{ALWAYS}(\neg(\mathit{AR\_command} \wedge \mathit{TSAFE\_command}))$
- *"All conflicts are eventually resolved"*
  $\text{ALWAYS}(\neg \mathit{TSAFE\_clear} \rightarrow \text{EVENTUALLY}\ \ \mathit{TSAFE\_clear})$
- *"All controller requests are eventually addressed"*
  $\text{ALWAYS}(\mathit{controller\_request} \rightarrow \text{EVENTUALLY}\ \ \neg \mathit{controller\_request})$
- *"All aircraft requests are eventually addressed"*
  $\text{ALWAYS}(\mathit{aircraft\_request} \rightarrow \text{EVENTUALLY}\ \ \neg \mathit{aircraft\_request})$

Formal Methods Give Absolute Assurance . . . at a Cost

There are great benefits!

- Ex: Model checking returns a counterexample trace or assurance one doesn't exist! [1]

Formal verification is hard:

- Theorem proving is undecidable.
- Model checking is intractable. (i.e., LTL model checking is PSPACE-complete.)

Can't we use something easier to achieve the same level of assurance?

- testing
- simulation
- fault-tolerance

[1] with some caveats, such as that M and φ are correct, valid

Verification Methods

- **Testing** proves the existence of the correct behavior for any given input
- **Simulation** scales testing and extends it to hypothetical systems
- **Fault tolerance** designs *resilient* systems that fail safely

ALL of these methods are invaluable for producing robust systems

So why do we ALSO need formal methods?

The Donald == Donald Knuth

"...
Beware of bugs in the above code;
I have only proved it correct, not tried it."
– Donald Knuth

`https://staff.fnwi.uva.nl/p.vanemdeboas/knuthnote.pdf`

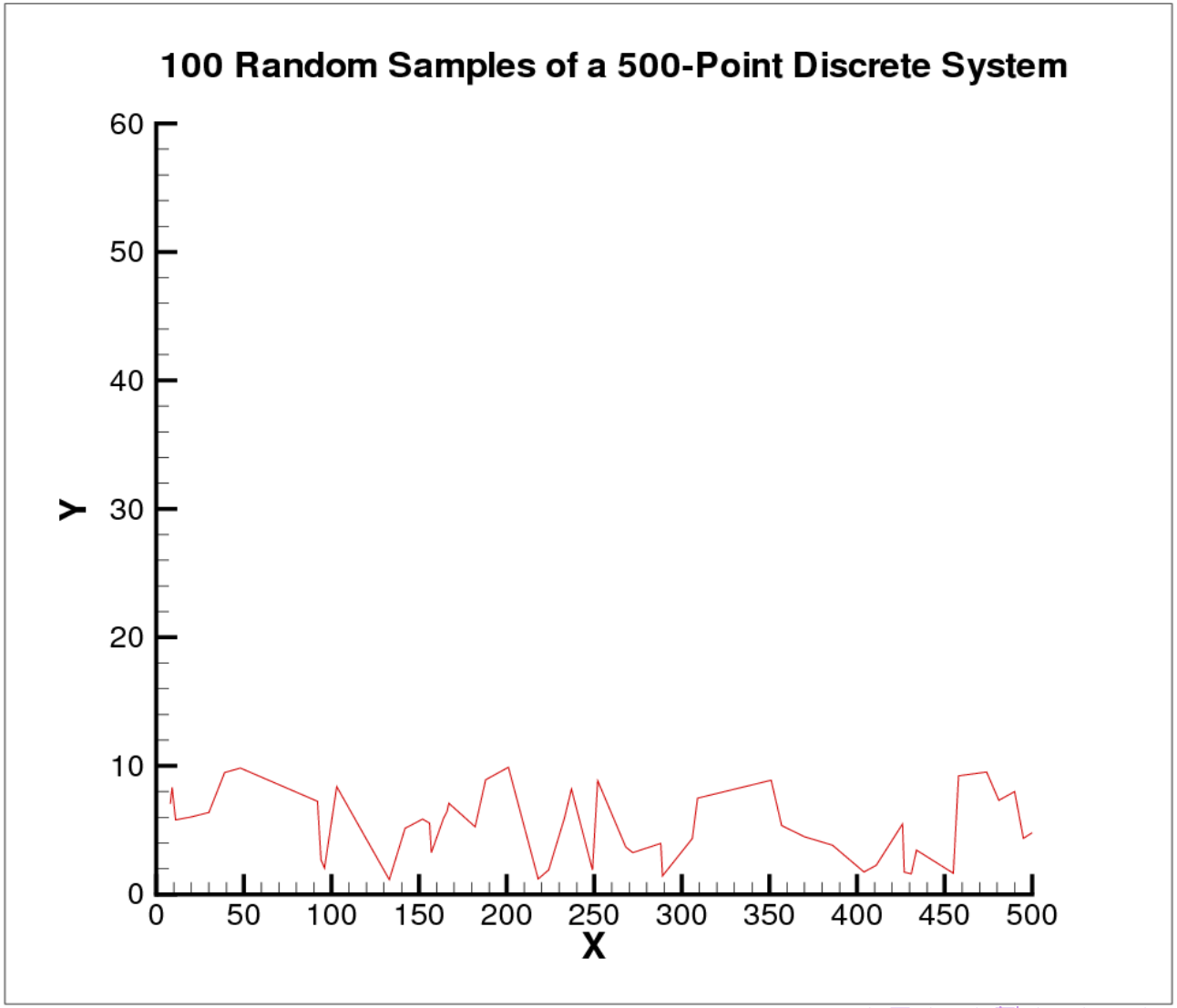
Hint: they solve different problems ...

The Random Testing Approach Does Not Work . . .
for Cyber-Physical or Software Systems

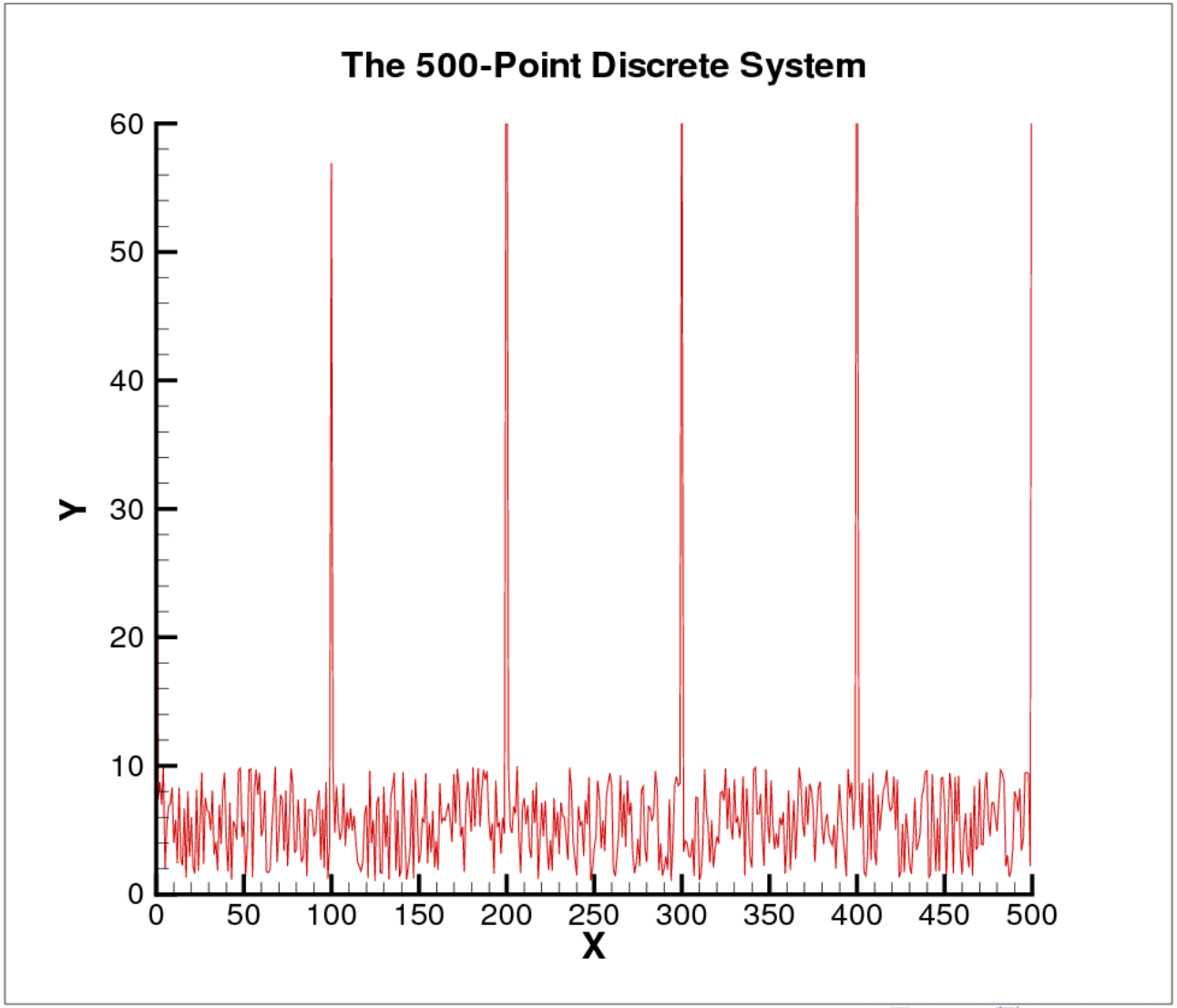
Random testing: Generate random trajectories through the state space. Estimate the probability that the property holds to a certain level of confidence.

- There is no reason why if something works for two data points, that it will work for data points in between.
- To discover a bug: have to get lucky and guess the right test case.
- Have to cover *every* data point to eliminate uncertainty.
- In software, each point is its own boundary (can't just check the boundary conditions).

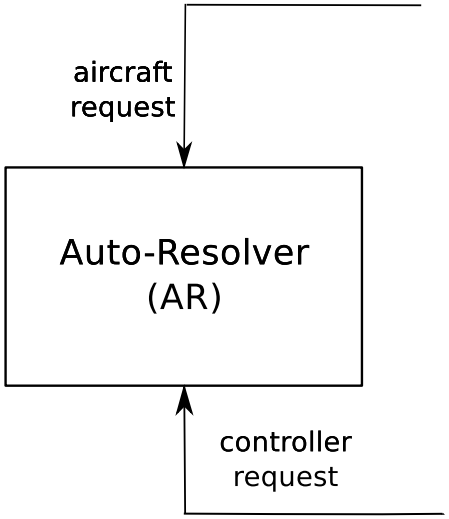
An Example . . . Traffic to a Website



Example Continued



Code Example: Request → Auto-Resolver



Specification:
Only one request is registered at a time.

```
global Boolean processing_request = 0;

function make_request(request) {
  //Only 1 request is allowed at a time
  if (processing_request == 1) {
    //Already processing a request
    return "request ignored";
  } //end if

  processing_request++; //flag: we're processing
  //Examine the request
  response = check_request(request);
  //Request cannot overrule TSAFE commands
  if ((TSAFE_command == 0)
    && (processing_request == 1)) {
    processing_request--; //unset flag
    return response;
  } //end if
} //end function
```

Property Violation

aircraft request and controller request interleave

```
if (processing_request == 1) { return "request ignored"; }
if (processing_request == 1) { return "request ignored"; }

processing_request++;
processing_request++;      processing_request is now 2!

response = check_request(request);
if ((TSAFE_command == 0) && (processing_request == 1)) { FAIL!
  processing_request--; //unset flag
  return response;
} //end if

response = check_request(request);
if ((TSAFE_command == 0) && (processing_request == 1)) { FAIL!
  processing_request--; //unset flag
  return response;
} //end if
```

DEADLOCK

Can't We Just Employ Fault-Tolerant Designs?

Fault-tolerant designs allow systems to continue operating in the presence of faults.

For *hardware* systems, fault-tolerance is achieved using redundant components, voting strategies, physical isolation, and algorithmic filtering.

- Hardware fault tolerance is most successful in recovering from *physical failures*.

For *software* systems, fault-tolerance is achieved by having independent programming teams create several versions of the software from the same system specification. Voting resolves any conflicts.

- The idea is to count on separate, redundant copies to fail independently.

Fault Tolerance $\neq$ Software Reliability

1. Redundant software versions cannot be proven to be independent.[2]
2. Multiple redundant copies are created from the same requirements. Faulty requirements mean *all* copies are also faulty. *Formal methods can debug requirements...*
3. Programmers make the same mistakes, even when programming independently. [3]
4. Redundancy adds complexity and complications in designing the voting algorithm and strategies for independent development.

[2] Ricky W. Butler and George B. Finelli. The Infeasibility of Quantifying the Reliability of Life-Critical Real-Time Software.

[3] John C. Knight, Nancy G. Leveson. An Experimental Evaluation of the Assumption of Independence in Multiversion Programming.

Why Don't We Verify Every Safety-Critical System at NASA?

1. Systems are not designed for verification.
2. Arms race: tools must scale to handle every system that is.

These problems are ranked in order! Problem #1 is the biggest obstacle to formal verification at NASA.

Black Boxes Cannot Be Formally Verified

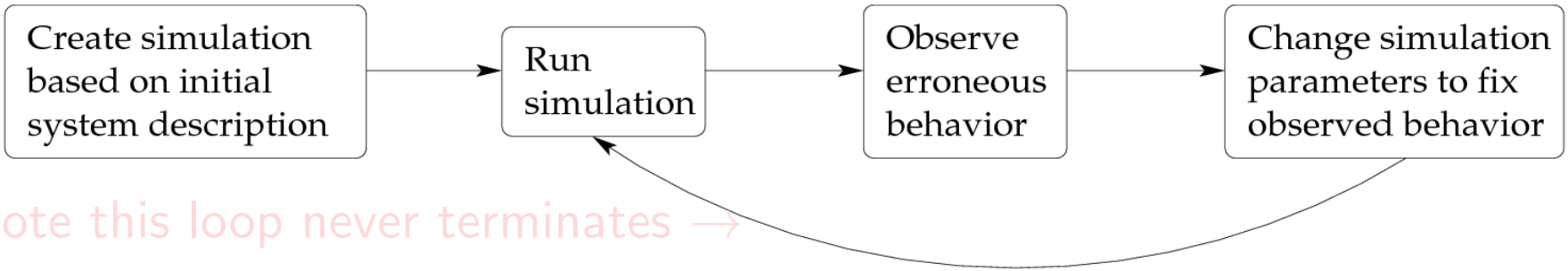
Formal Methods involves logic, not magic.

- Recall that model checking asks if system model M satisfies (models) property φ.
- If you don't know exactly what it does and what it's supposed to do, how can you tell that it does exactly what it is supposed to?

Problem 1: Systems are not designed for verification.

Heuristics Are Not a Good Design Method for Reliable Software

Heuristic Software Development Cycle:



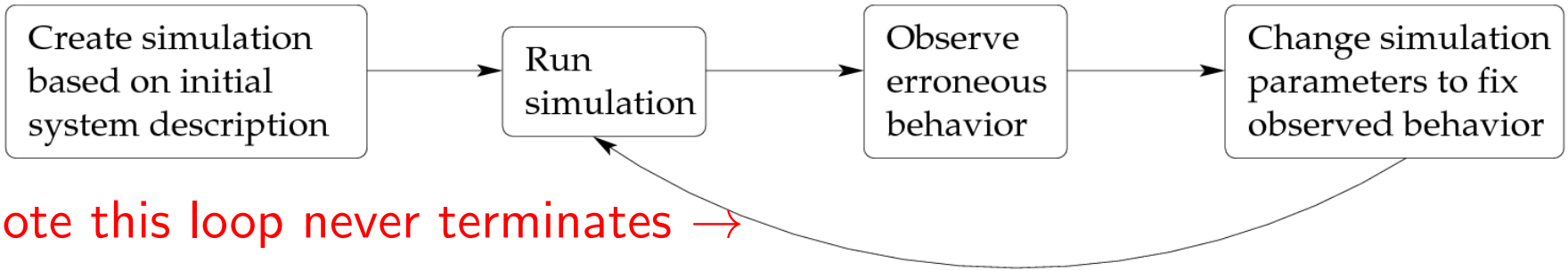
- Continually tweaking the system to fix faulty behaviors results in an algorithm based on special cases.
- Simulation results are necessarily bounded by confidence intervals due to reliance on a finite set of trajectories.

In the end, do you really know what the system does for any input?

Problem 1: Systems are not designed for verification.

Heuristics Are Not a Good Design Method for Reliable Software

Heuristic Software Development Cycle:



Note this loop never terminates →

- Continually tweaking the system to fix faulty behaviors results in an algorithm based on special cases.
- Simulation results are necessarily bounded by confidence intervals due to reliance on a finite set of trajectories.

In the end, do you really know what the system does for any input?

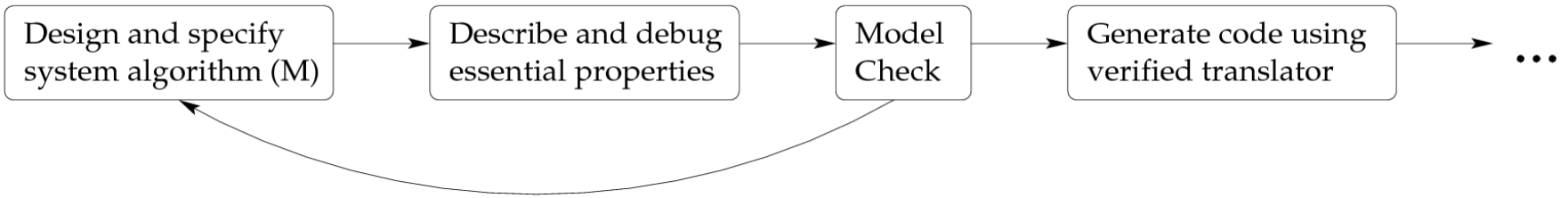
Some Systems Shouldn't Be Formally Verified

Formal verification is actually *undesirable* for some systems:

- Systems where failure is not costly.
- Systems where occasional failure is beneficial.
  - Would you buy Software Version 2.0 if Version 1.0 worked perfectly?
- Systems where we necessarily have inaccurate/unbounded information or quantifiable uncertainty.
  - Biological systems are already imperfect models of reality due to limited understanding.
- Systems whose structures aren't suited to formal specification.
  - Natural language processing: natural languages are context-sensitive and therefore cannot be represented by automata. Natural languages are ambiguous by nature.

Solution: Design from the Start for Verification

A Better Software Development Cycle:



- A system can only be verified if you know what it does and why it should work.
- Formal verification requires visibility into design details.

Formal methods can be used *in the design phase* to enhance understanding and avoid logical flaws in algorithm/specification designs.

Formal Verification Success Stories

Formal verification techniques have been successfully used for a wide variety of real systems:

- Aerospace:
  - Air traffic control i.e. Small Aircraft Transportation System (SATS)
  - KB3D, TCAS: algorithms for 3-D conflict detection and resolution
  - Java pathfinder: verify executable Java bytecode (on Mars rovers!)
- Cars:
  - Self-driving cars: Toyota, Mitre, others
  - Toyota Prius court case
- Intel chips:
  - floating-point mathematical functions and other properties of hardware designs
  - post-silicon revisions to fix bugs found at silicon test
- Protocols: TCP/IP, communication protocols
- Microsoft device drivers (e.g., SLAM)
- Program termination and liveness (via Terminator)

Solutions: Mitigate the State Explosion Problem

Model Checking is largely automated and gives *counterexamples*.
Theorem proving is well-suited to reasoning about *very large state spaces*.

State explosion problem: state spaces of real systems can be very large, even infinite. This is the biggest challenge for model checking.

Mitigation:

- Abstractions that capture all relevant aspects of the system design but reduce the state space.
- Data structures that conserve memory: efficient hashing, favorable BDD variable ordering, etc.
- Component-based verification that logically divides the system into smaller components (good for parallelization!)

Challenges and Directions for Future Research

- Time vs. Space: Real systems have large state-spaces. It takes time to fill large quantities of memory.
- Writing verification tools is hard. Many tools are not mature enough for industrial use. Others are not widely available.
- Tools are not fully automatic:
  - Theorem Proving requires heavy user guidance.
  - Model Checking requires knowledge of temporal logic, specification strategies, and abstraction techniques.
- Verified specification/code translators are still primitive, do not handle complex code structures like objects, templates, inheritance, etc. *Synthesis is taking off!*

Challenges and Directions for Future Research

- Time vs. Space: Real systems have large state-spaces. It takes time to fill large quantities of memory.
- Writing verification tools is hard. Many tools are not mature enough for industrial use. Others are not widely available.
- Tools are not fully automatic:
  - Theorem Proving requires heavy user guidance.
  - Model Checking requires knowledge of temporal logic, specification strategies, and abstraction techniques.
- Verified specification/code translators are still primitive, do not handle complex code structures like objects, templates, inheritance, etc. *Synthesis is taking off!*

Questions?

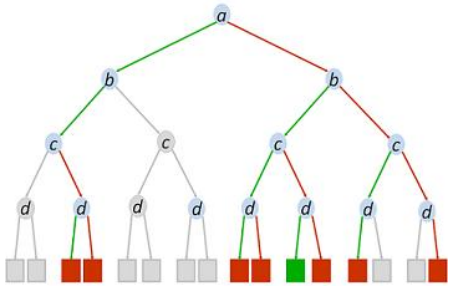
Problem 2: Arms race: tools must scale to handle every system that is.

Research Interests

Automated Reasoning

- Avionics/Flight Software
- Satisfiability (SAT)/SMT
- AI/Algorithms
- Explainability

Formal Specification

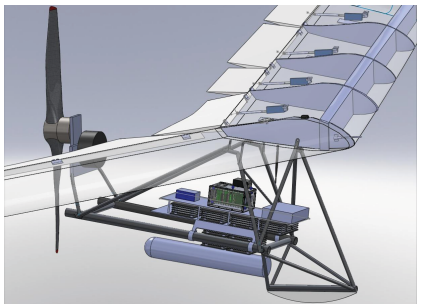


- Specification Patterns
- Specification Debugging
- Consistency/Temporal Satisfiability Checking

Design-time Safety Analysis

- Model Checking (Explicit and Symbolic)
- Model Based Design
- Requirements Elicitation
- Temporal Logic Encoding

Runtime Verification



- R2U2 Engine
- System Health Management
- Resource-limited Sanity Checking
- Automated Diagnostics/Prognostics
- Real-time Intelligent Sensor Fusion