UNIVERSITY OF WATERLOO

CS 856: Programmable Networks

Lecture 7: Network Verification

Mina Tahmasbi Arashloo

Winter 2023

Logistics

- Project progress report is due **Friday, March 10th**
  - Two pages
  - Briefly describe the motivation and problem statement
  - Briefly describe the related work, including any new ones you have found since the proposal
  - Describe what you have achieved so far
  - Describe what you plan to do for the rest of the term
- Assignment two will be released next week and is optional (extra credit)

Formal Verification

Proving or disproving

the correctness of a (software or hardware) system

with respect to a certain formal specification or property

using formal methods of mathematics

Formal Verification

e.g., Traffic light controller

Proving or disproving

the correctness of a (software or hardware) system

with respect to a certain formal specification or property

using formal methods of mathematics

Formal Verification

e.g., Traffic light controller

Proving or disproving

the correctness of a (software or hardware) system

with respect to a certain formal specification or property

usi… al methods of mathematics

Safety properties:
nothing bad happens

e.g., traffic light should not be simultaneously green in both direction

Formal Verification

e.g., Traffic light controller

Proving or disproving

the correctness of a (software or hardware) system

with respect to a certain formal specification or property

usi al methods of ma atics

Safety properties:
nothing bad happens

e.g., traffic light should not be simultaneously green in both direction

Liveness properties:
something good eventually happens

e.g., If there is a car on the road, the light will eventually turn green

Formal Verification

Broadly applicable!

- Hardware design
- Software
- Distributed systems
- **Computer Networks**
- Aviation
- Neural Networks
- …

Proving o

the correctness of a (sof

with respect to a certain formal specification or property

using formal methods of mathematics

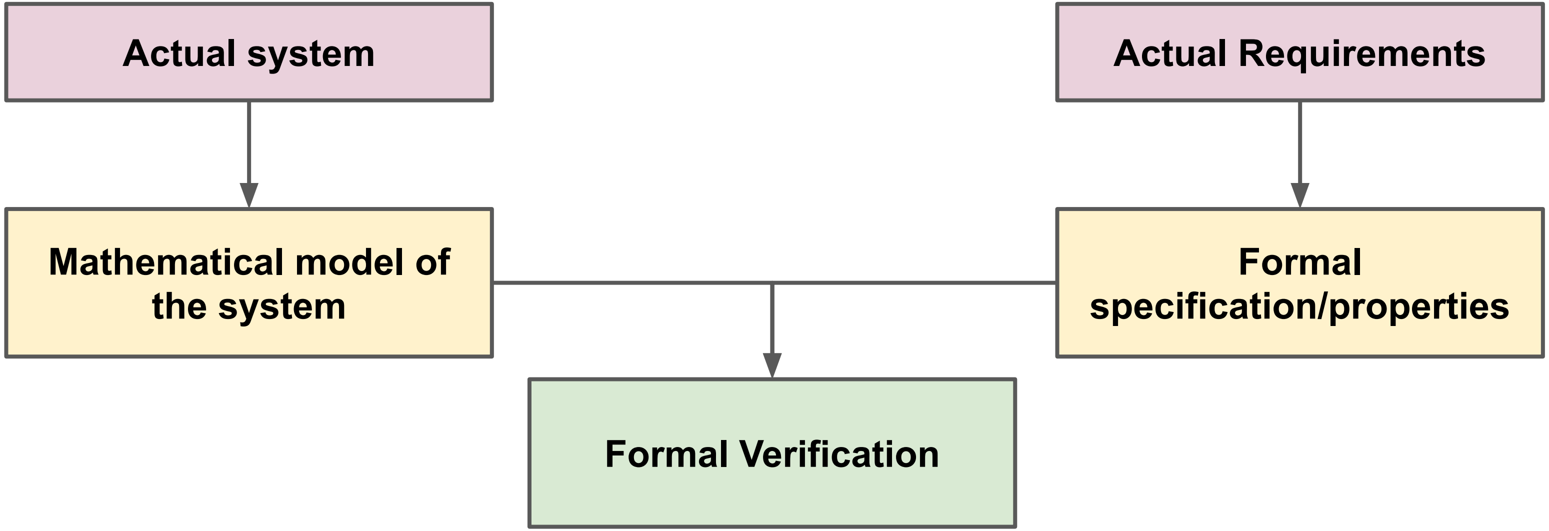
How do we go about verifying a system?

Actual system

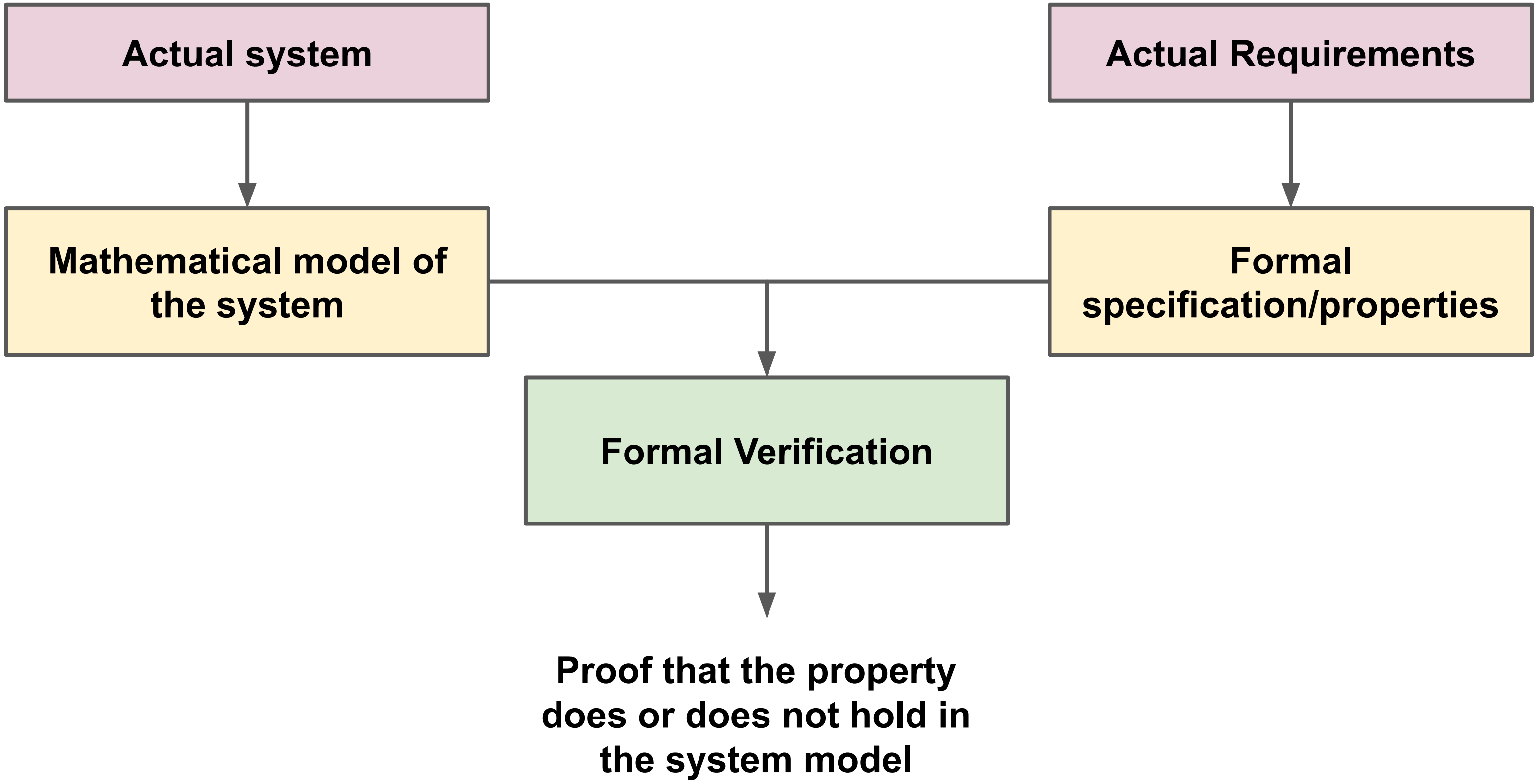
Actual Requirements

How do we go about verifying a system?

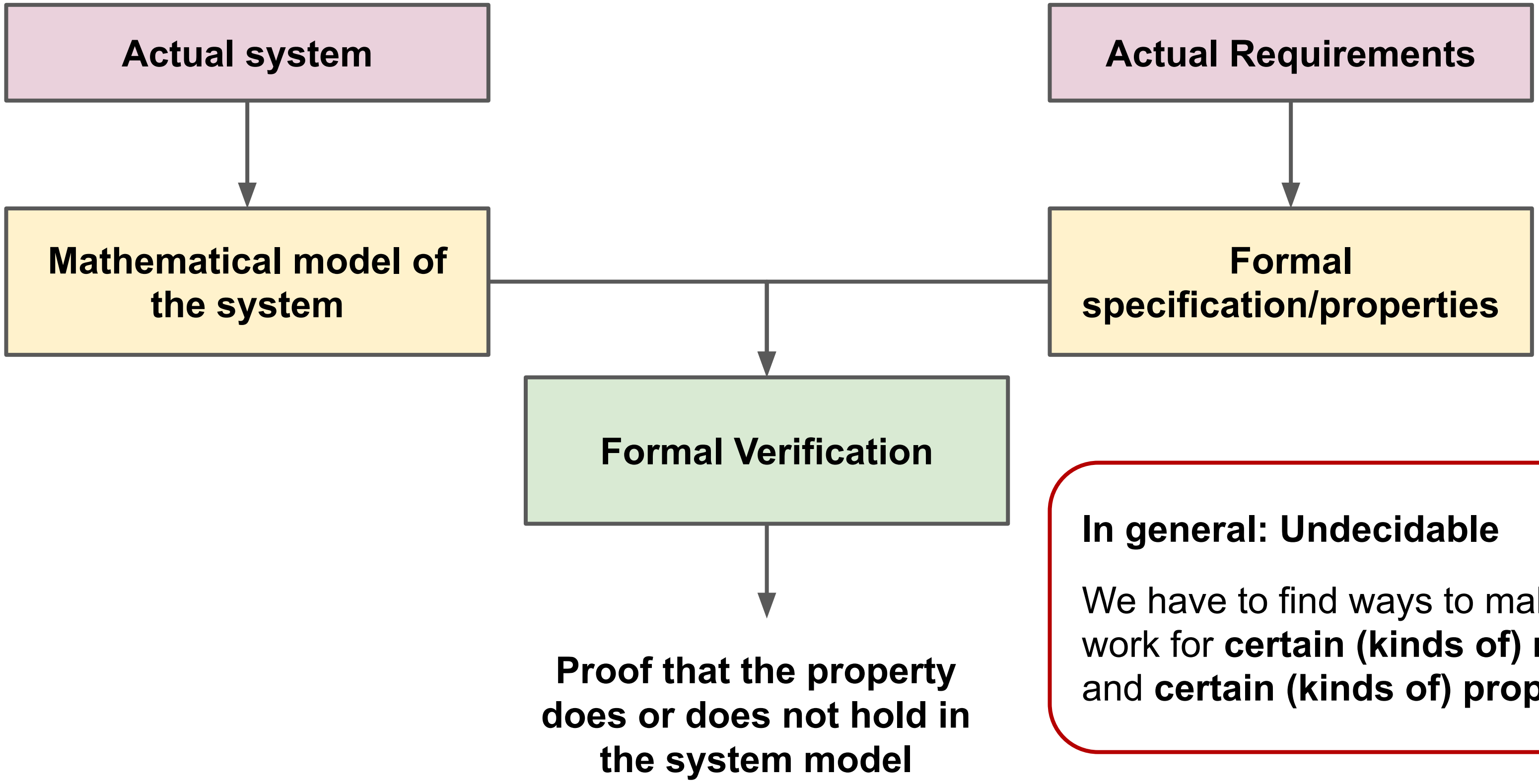
How do we go about verifying a system?



How do we go about verifying a system?



How do we go about verifying a system?



In general: Undecidable

We have to find ways to make it work for **certain (kinds of) models** and **certain (kinds of) properties**

A (very) simple example

The following example is adapted from Aarti Gupta's Fall'15 course on "Automated Reasoning about Software" at Princeton University

A (very) simple example

```
bool x;
int y = 8;
int z = 0;
int w = 0;
if (x)
   z = y - 1
else
   w = y + 1

assert(z == 5
|| w == 9)
```

A (very) simple example

Actual program

```
bool x;
int y = 8;
int z = 0;
int w = 0;
if (x)
   z = y - 1
else
   w = y + 1
```

```
assert(z == 5
|| w == 9)
```

Actual requirements

A (very) simple example

Actual program

```
bool x;
int y = 8;
int z = 0;
int w = 0;
if (x)
   z = y - 1
else
   w = y + 1
```

```
assert(z == 5
|| w == 9)
```

Actual requirements

Mathematical model (in logic) of the program

$$(y = 8) \wedge$$

$$(x \rightarrow (z = y - 1)) \wedge$$
$$(\neg x \rightarrow (z = 0)) \wedge$$

$$(x \rightarrow (w = 0)) \wedge$$
$$(\neg x \rightarrow (w = y + 1))$$

A (very) simple example

Actual program

```
bool x;
int y = 8;
int z = 0;
int w = 0;
if (x)
    z = y - 1
else
    w = y + 1

assert(z == 5
|| w == 9)
```

Actual requirements

Mathematical model (in logic) of the program

$$(y = 8) \land$$

$$(x \rightarrow (z = y - 1)) \land$$
$$(\neg x \rightarrow (z = 0)) \land$$

$$(x \rightarrow (w = 0)) \land$$
$$(\neg x \rightarrow (w = y + 1))$$

Formal property

$$(z = 5) \,||\, (w = 9)$$

A (very) simple example

Actual program

```
bool x;
int y = 8;
int z = 0;
int w = 0;
if (x)
   z = y - 1
else
   w = y + 1
```

```
assert(z == 5
|| w == 9)
```

Actual requirements

Mathematical model (in logic) of the program

$(y = 8) \wedge$

$(x \rightarrow (z = y - 1)) \wedge$
$(\neg x \rightarrow (z = 0)) \wedge$

$(x \rightarrow (w = 0)) \wedge$
$(\neg x \rightarrow (w = y + 1))$

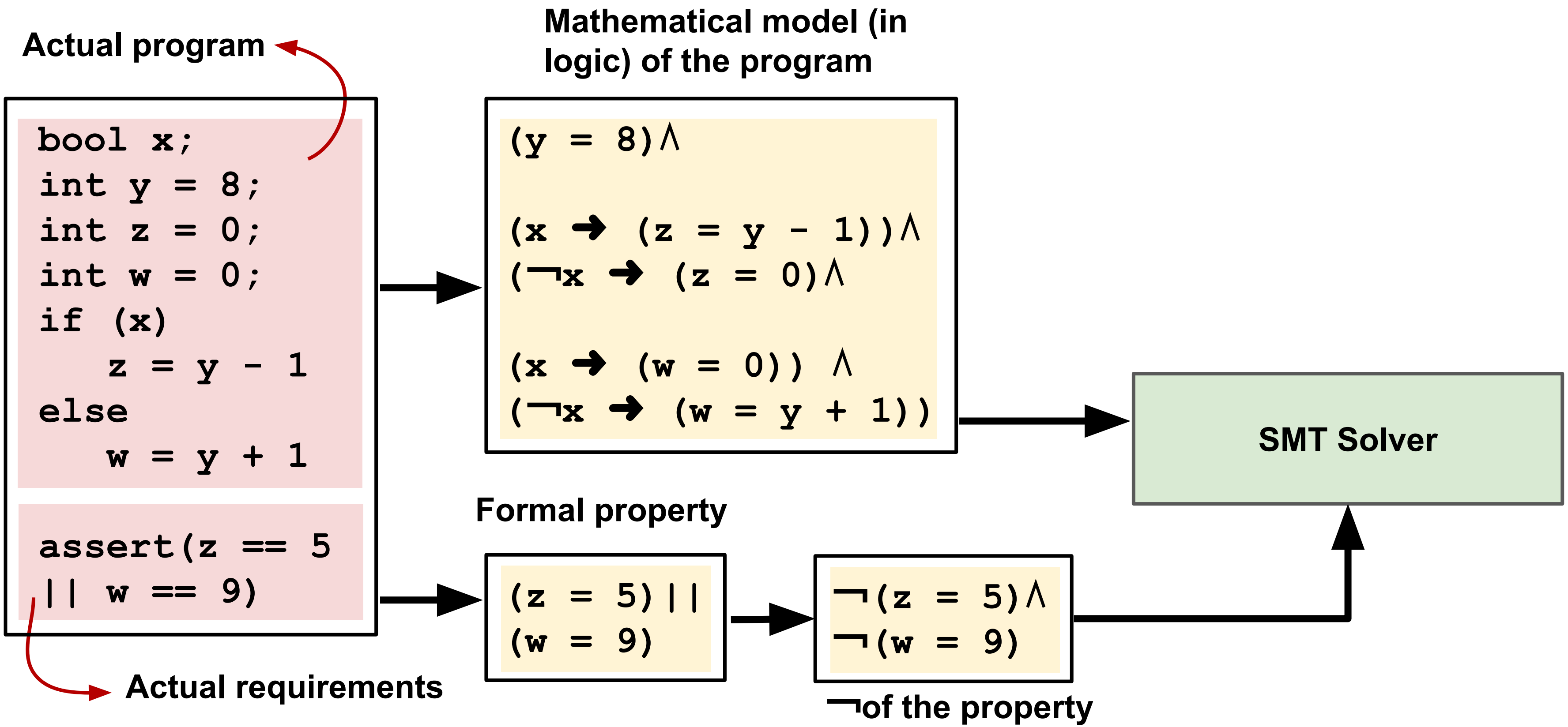
Formal property

$(z = 5) \,||\,$
$(w = 9)$

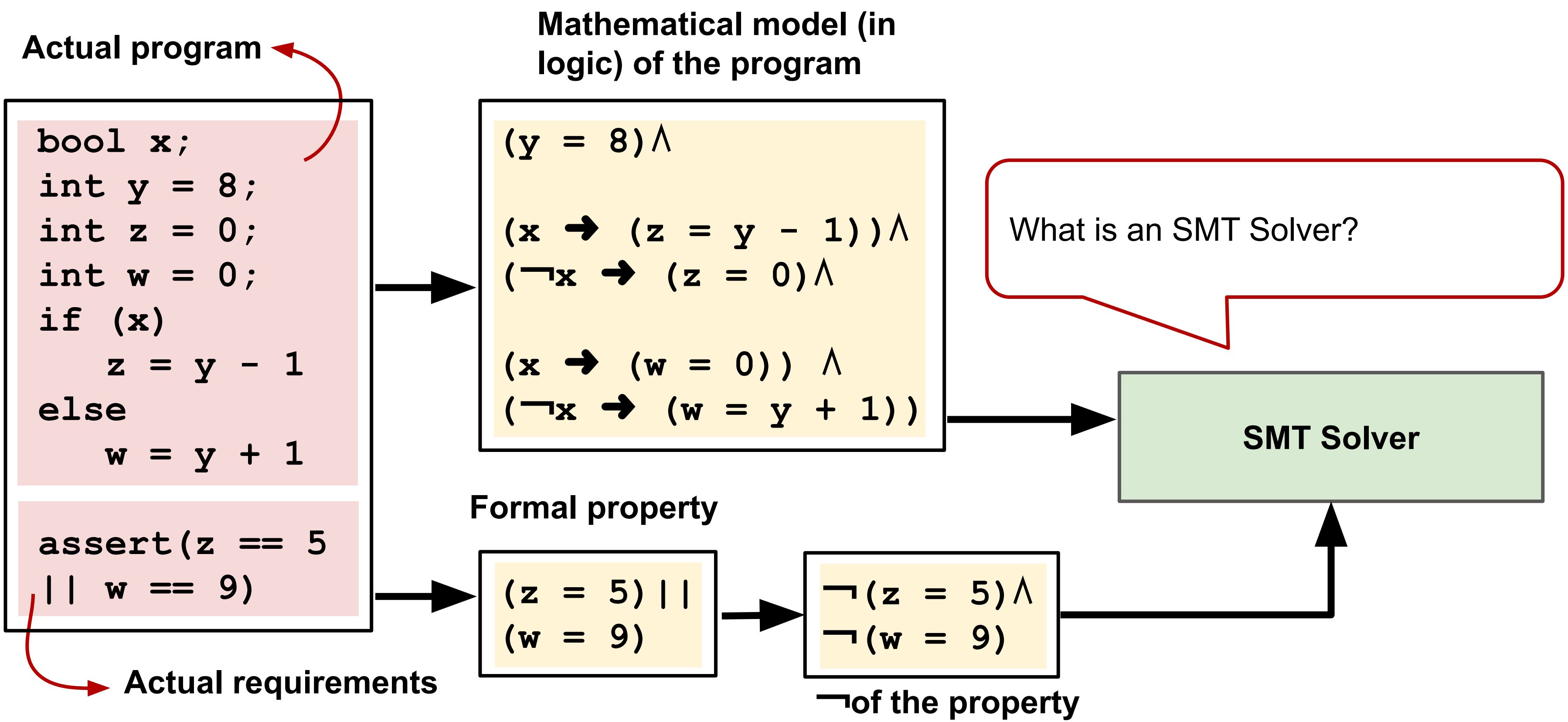
$\neg(z = 5) \wedge$
$\neg(w = 9)$

$\neg$ of the property

A (very) simple example



A (very) simple example



Satisfiability Modulo Theories (SMT)

- Let's look at the boolean satisfiability problem (SAT) first.

The (Boolean) Satisfiability Problem (SAT)

- Suppose you have a boolean formula
  - e.g., $(a \lor b) \land (\neg b \lor c)$
- You can assign true or false to each variable
- Is there an assignment that will make the entire formula evaluate to true?
- This is the SAT problem
- In general, it is NP complete
  - Unless P = NP, it can't be solved in polynomial time

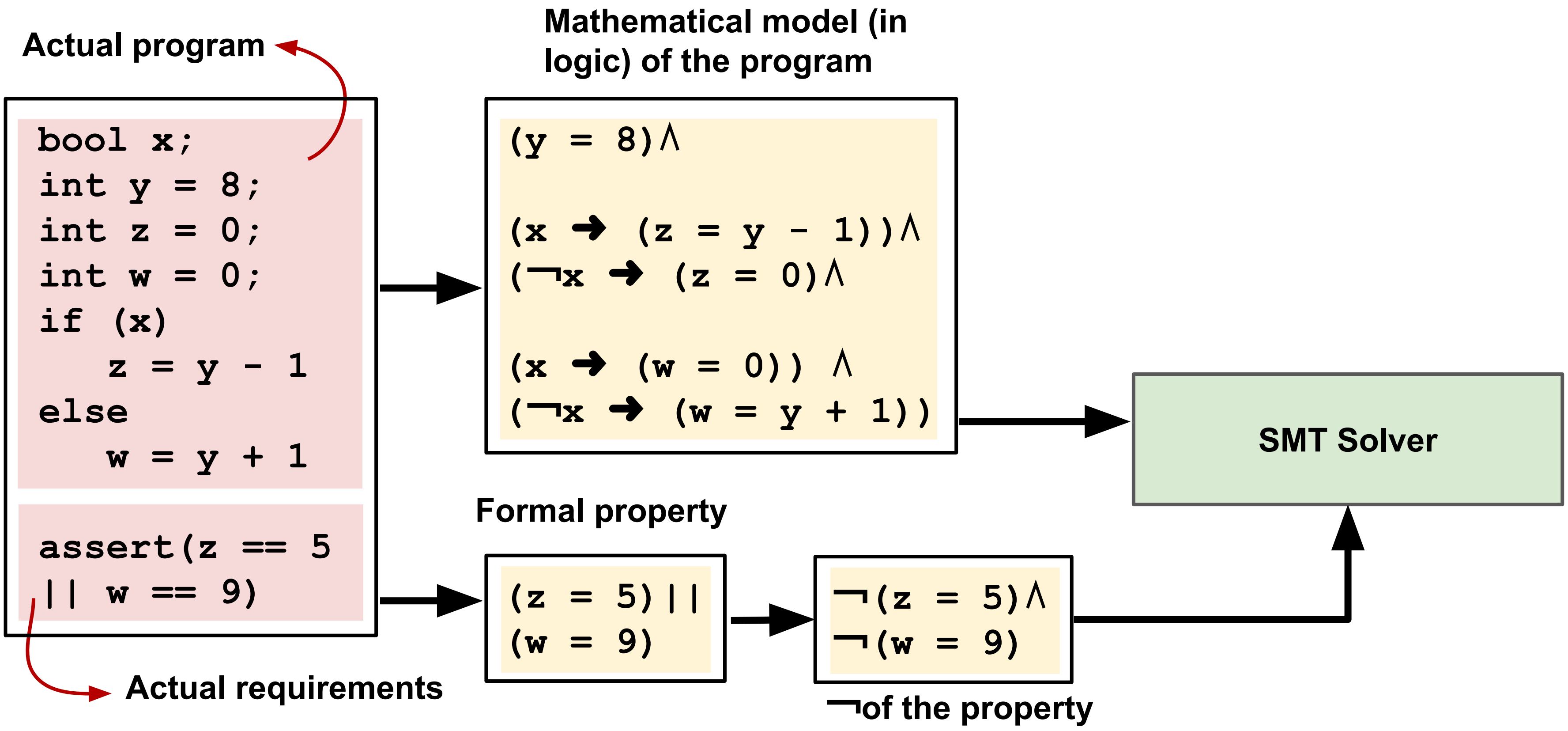
The (Boolean) Satisfiability Problem (SAT)

- The SAT problem, in general, is NP complete
  - Unless P = NP, it can't be solved in polynomial time
- Still, in the formal methods community, there has been a significant progress in tools that can, in many cases, solve this problem quite quickly for large formulas.

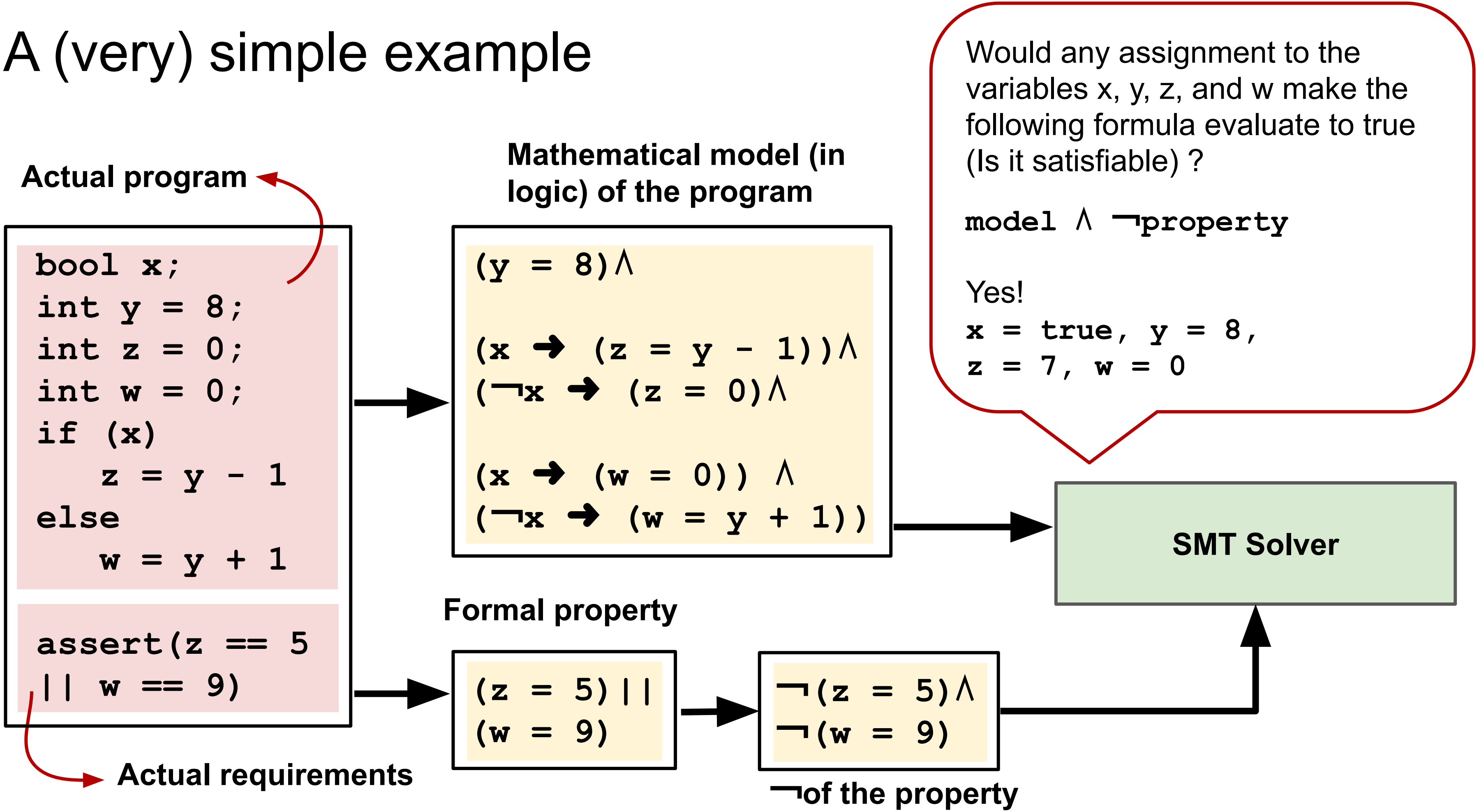
Satisfiability Modulo Theories (SMT)

- The same satisfiability problem, but for more complex (first-order-logic) formulas
  - integer variables, real variables, …
  - arrays, bit vectors, lists, strings, …
  - functions such as equality, addition, subtraction, …
- Harder problem
  - can be NP-hard or undecidable depending on the "theory"
- but we have found ways to make it work by finding algorithms for analyzing certain families of formulas ("theories").

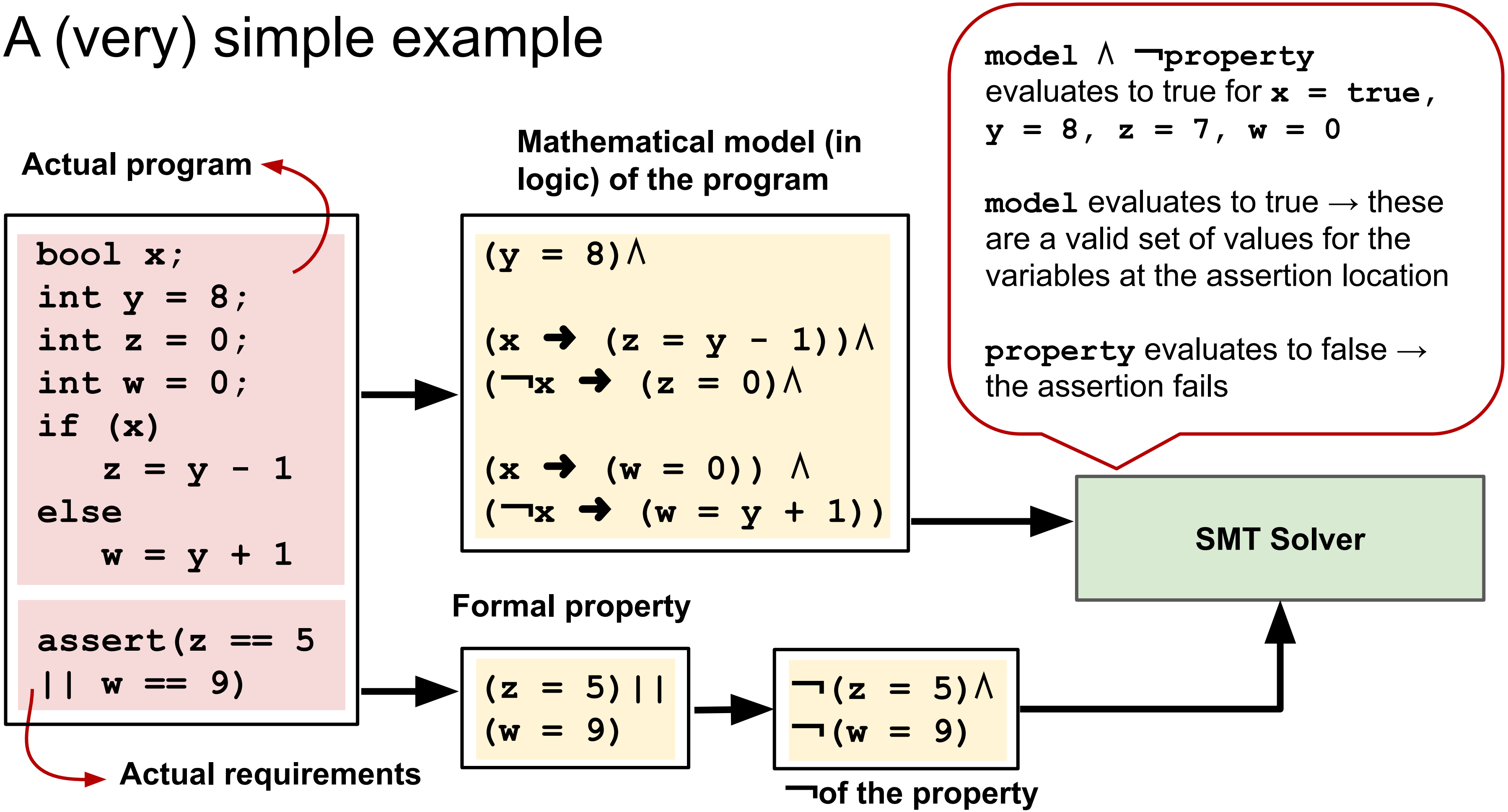
A (very) simple example

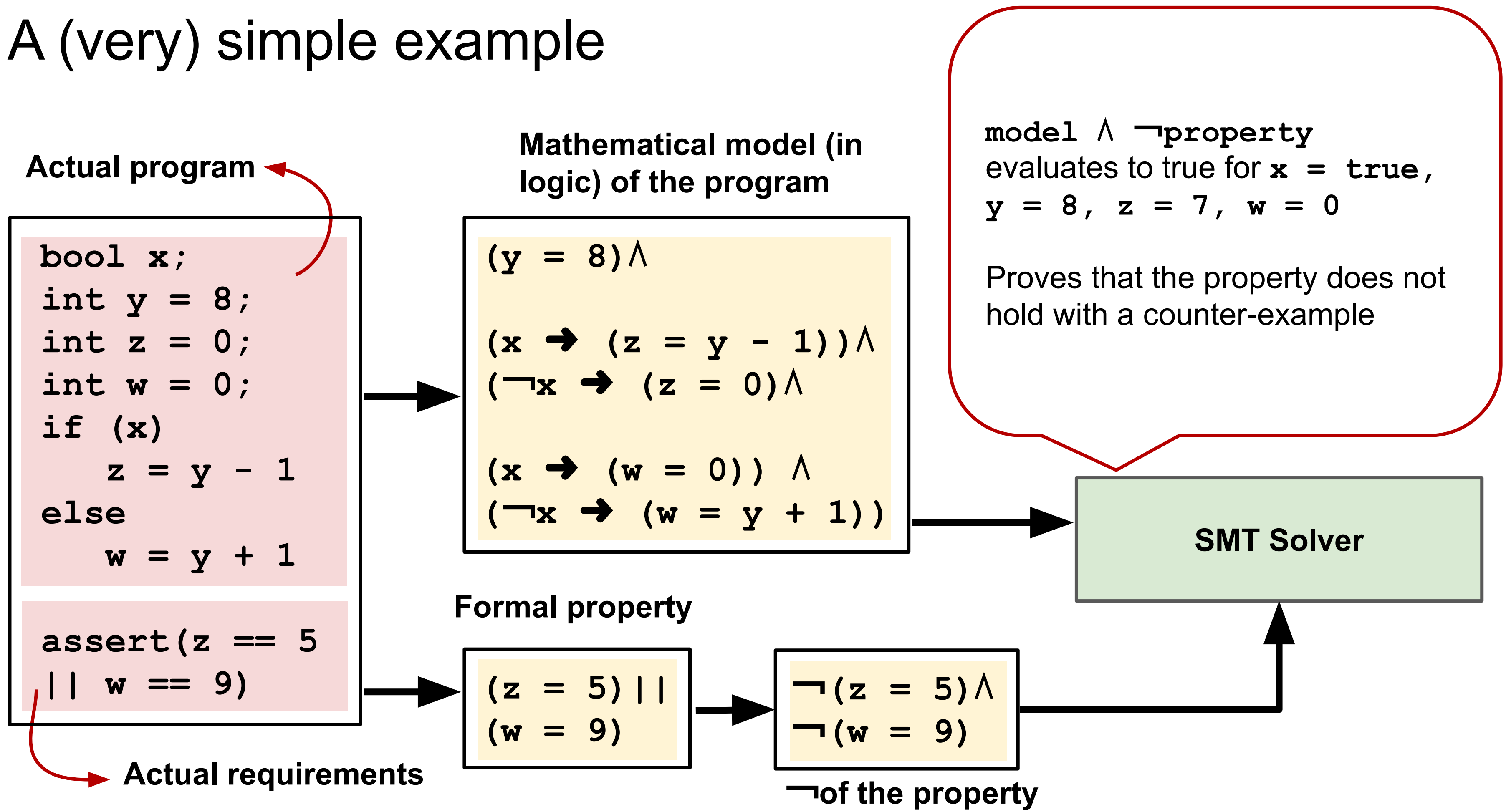


A (very) simple example



A (very) simple example




A (very) simple example
Actual program
bool x;
int y = 8;
int z = 0;
int w = 0;
if (x)
z = y - 1
else
w = y + 1
assert(z == 5
|| w == 9)
Actual requirements
Mathematical model (in logic) of the program
(y = 8)∧
(x ➜ (z = y - 1))∧
(¬x ➜ (z = 0)∧
(x ➜ (w = 0)) ∧
(¬x ➜ (w = y + 1))
Formal property
(z = 5)||
(w = 9)
¬(z = 5)∧
¬(w = 9)
¬of the property
SMT Solver
model ∧ ¬property
evaluates to true for x = true,
y = 8, z = 7, w = 0
Proves that the property does not hold with a counter-example

A (very) simple example

Actual program

```
bool x;
int y = 8;
int z = 0;
int w = 0;
if (x)
   z = y - 1
else
   w = y + 1
```

```
assert(z == 5
|| w == 9)
```

Actual requirements

A (very) simple example

Actual program

```
bool x;
int y = 8;
int z = 0;
int w = 0;
if (x)
    z = y - 1
else
    w = y + 1

assert(z == 5
|| w == 9)
```

Let's change this to 7

Actual requirements

A (very) simple example

Actual program

```
bool x;
int y = 8;
int z = 0;
int w = 0;
if (x)
   z = y - 1
else
   w = y + 1

assert(z == 7
|| w == 9)
```

Actual requirements

A (very) simple example

Actual program

```
bool x;
int y = 8;
int z = 0;
int w = 0;
if (x)
   z = y - 1
else
   w = y + 1
```

```
assert(z == 7
|| w == 9)
```

Actual requirements

Mathematical model (in logic) of the program

```
(y = 8)∧

(x ➜ (z = y - 1))∧
(¬x ➜ (z = 0)∧

(x ➜ (w = 0)) ∧
(¬x ➜ (w = y + 1))
```

A (very) simple example

Actual program

```
bool x;
int y = 8;
int z = 0;
int w = 0;
if (x)
   z = y - 1
else
   w = y + 1
```

```
assert(z == 7
|| w == 9)
```

Actual requirements

Mathematical model (in logic) of the program

$(y = 8) \land$

$(x \rightarrow (z = y - 1)) \land$
$(\neg x \rightarrow (z = 0)) \land$

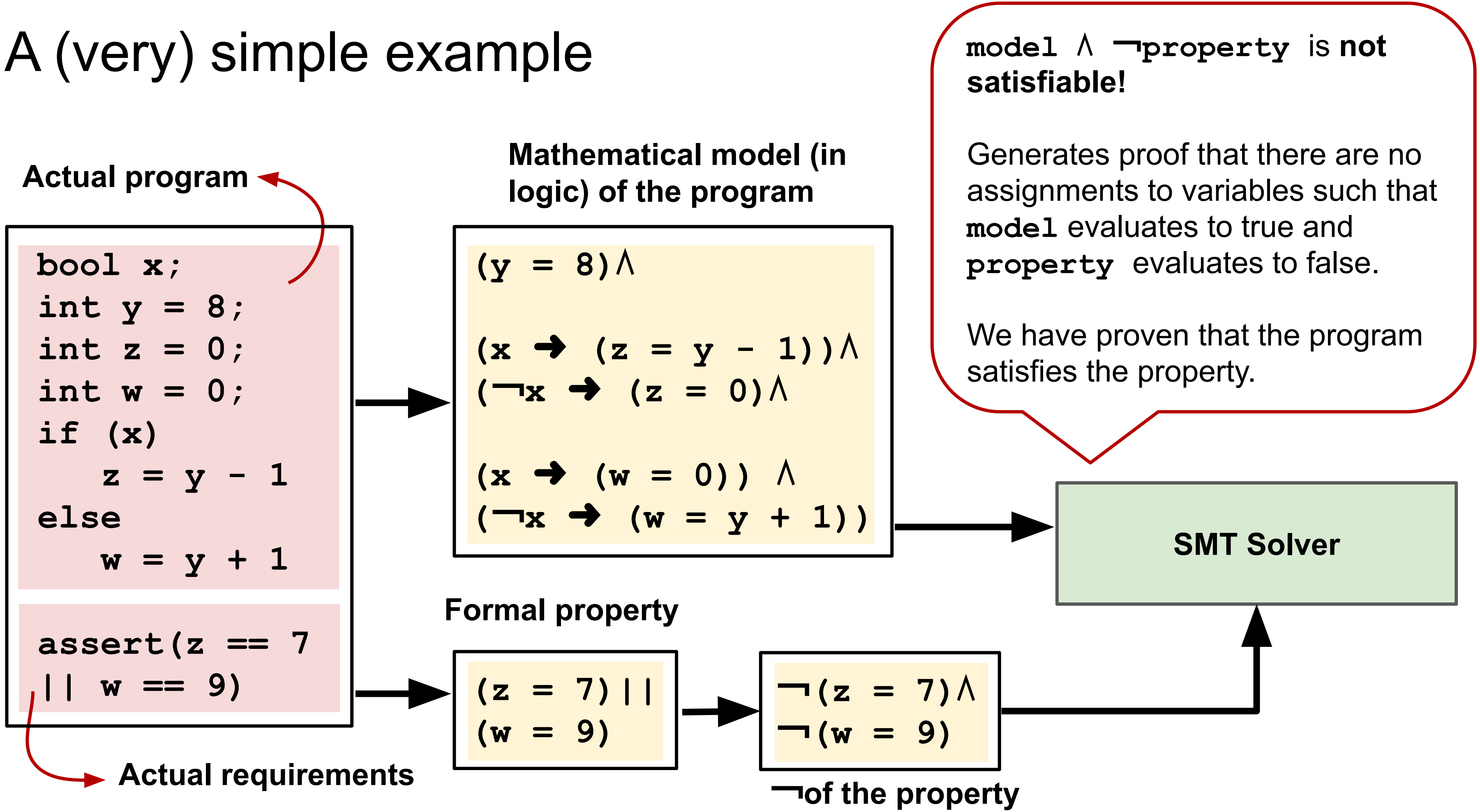
$(x \rightarrow (w = 0)) \land$
$(\neg x \rightarrow (w = y + 1))$

Formal property

$(z = 7) \,||\, (w = 9)$

$\neg(z = 7) \land \neg(w = 9)$

¬of the property


A (very) simple example
Actual program
bool x;
int y = 8;
int z = 0;
int w = 0;
if (x)
z = y - 1
else
w = y + 1
assert(z == 7
|| w == 9)
Actual requirements
Mathematical model (in logic) of the program
(y = 8)∧
(x ➜ (z = y - 1))∧
(¬x ➜ (z = 0)∧
(x ➜ (w = 0)) ∧
(¬x ➜ (w = y + 1))
Formal property
(z = 7)||
(w = 9)
¬(z = 7)∧
¬(w = 9)
¬of the property
SMT Solver
model ∧ ¬property is not satisfiable!
Generates proof that there are no assignments to variables such that model evaluates to true and property evaluates to false.
We have proven that the program satisfies the property.

What we haven't talked about (and won't) in this lecture …

- Kripke structures
- Temporal logic
- model checking
- symbolic execution
- Binary Decision Diagrams (BDD)
- Synthesis
- …

What we haven't talked about (and won't) in this lecture ...

- Kripke structures
- Temporal logic
- model checking
- symbolic execution
- Binary Decision Diagra
- **Synthesis**
- …

Generating a "program" that satisfies a high-level formal specification

- Program synthesis
- Invariant synthesis
- compiler optimizations
- …

Many use cases networking to generate:

- packet processing code for programmable data planes
- configurations and configuration updates
- control-plane repairs
- …

Why use formal verification in networking?

- Networks are growing increasingly complex.
    - They can have hundreds or thousands of interacting components
    - The functionality running in each component is getting more complex
    - configurations files can grow as large as thousands of lines
- Networks are becoming a critical infrastructure
    - Bugs can take down the network or reduce its performance.
    - Network problems can affect thousands if not millions of people
- We need to catch bugs (or prove lack thereof) proactively before going into production

Formal verification in networking

- Started with verifying the forwarding properties of the data plane and control plane.

- Now expanding into more complex functionalities and properties
  - DNS, network performance, …

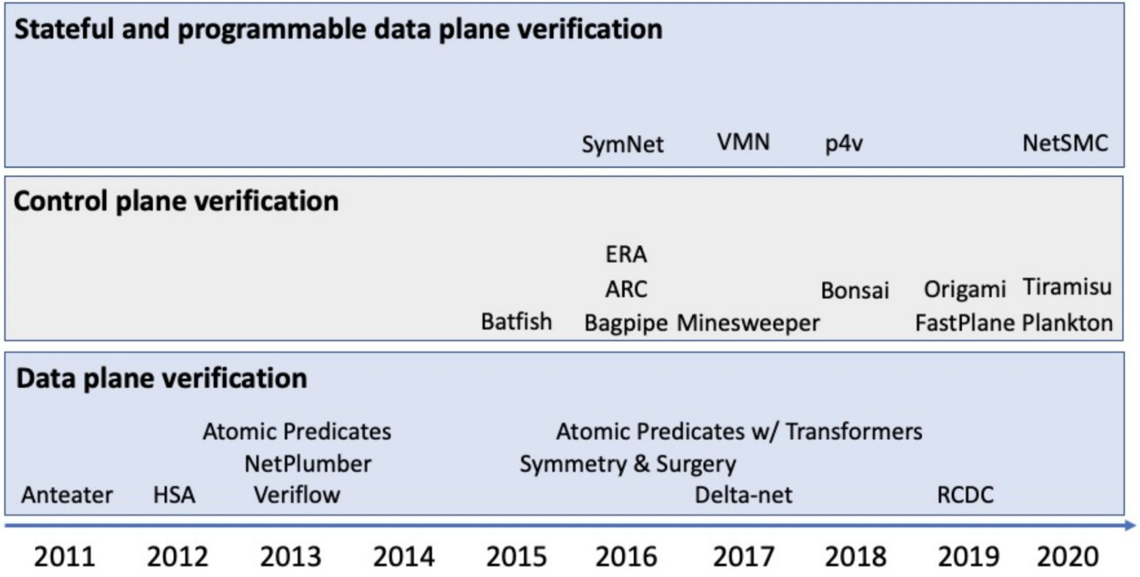


Figure taken from netverify.fun

Formal verification in networking

- Started with verifying the forwarding properties of the data plane and control plane.
- Now expanding into more complex functionalities and properties
  - DNS, network performance, …

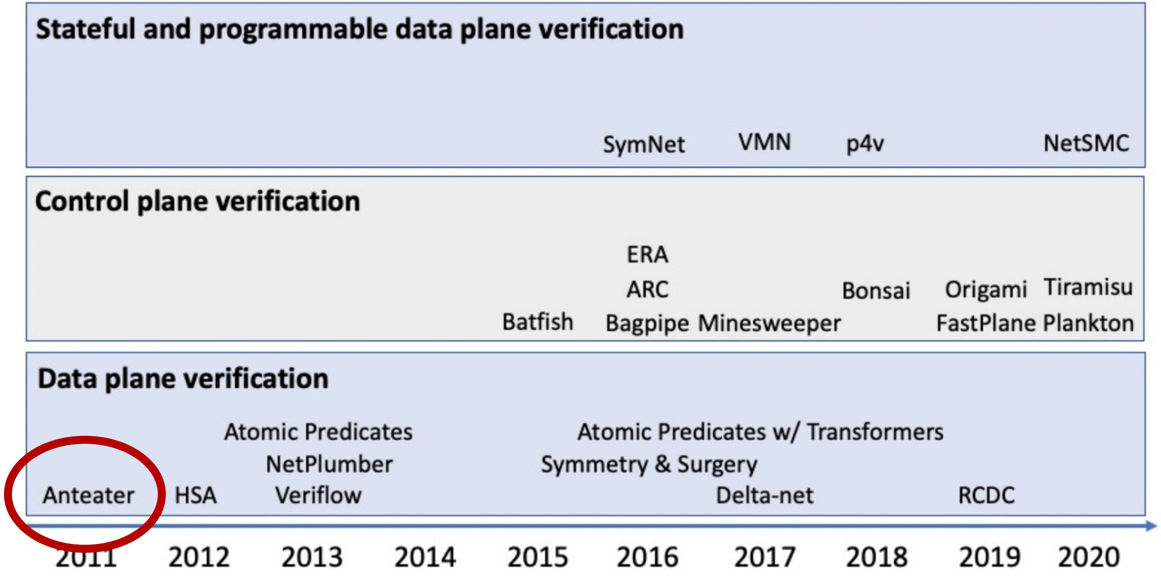
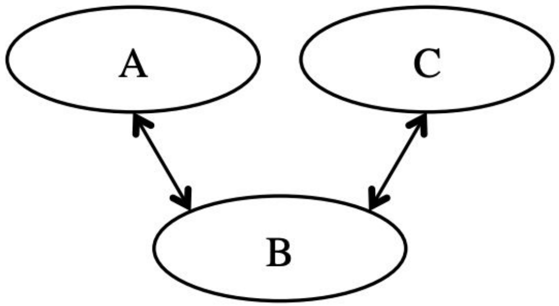


Figure taken from netverify.fun

Example - Anteater (SIGCOMM'11)

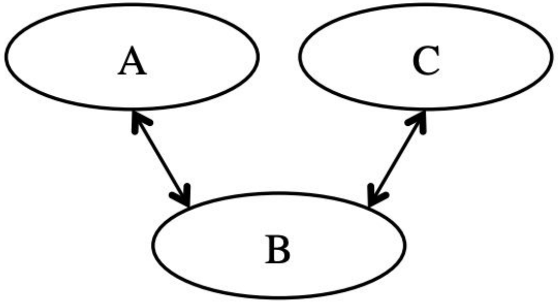
- Models the forwarding rule on the data plane as boolean formulas
- Uses a SAT solver to verify invariants about the network behavior
- The invariants are mostly related to forwarding
  - Reachability
  - Absence of forwarding loops
  - Absence of blackhols

Example - Anteater (SIGCOMM'11)

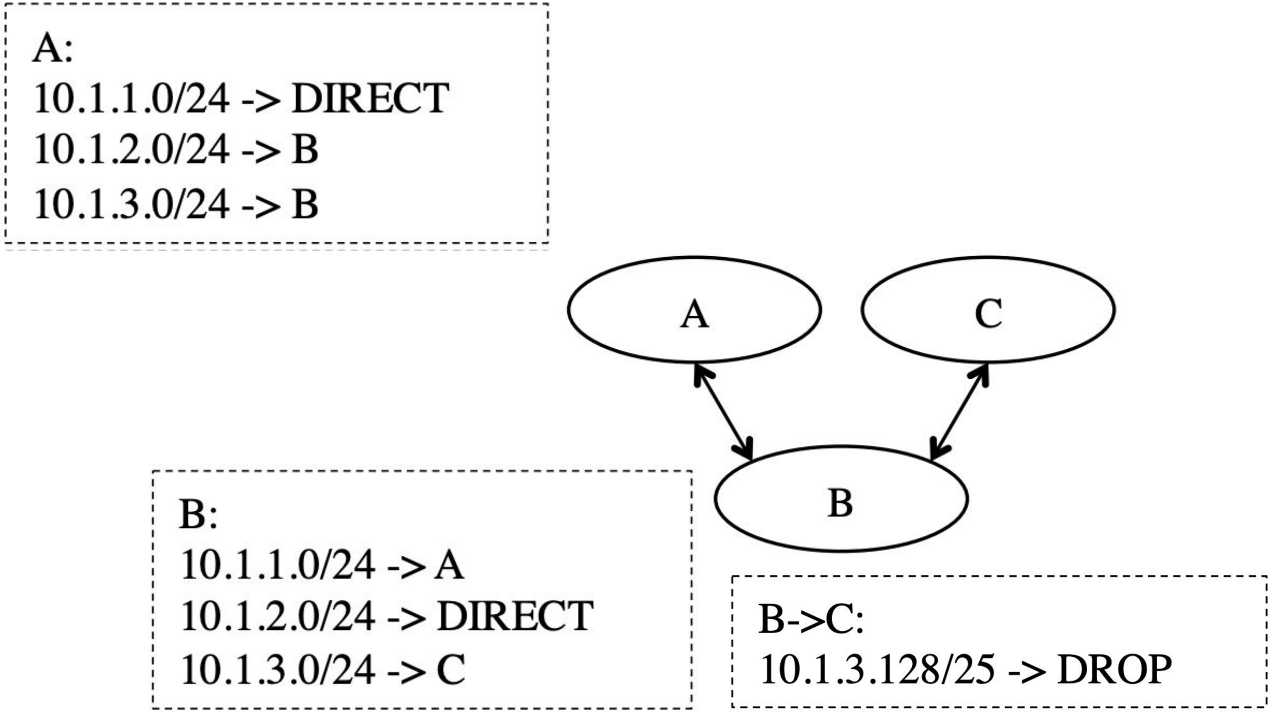


Example - Anteater (SIGCOMM'11)

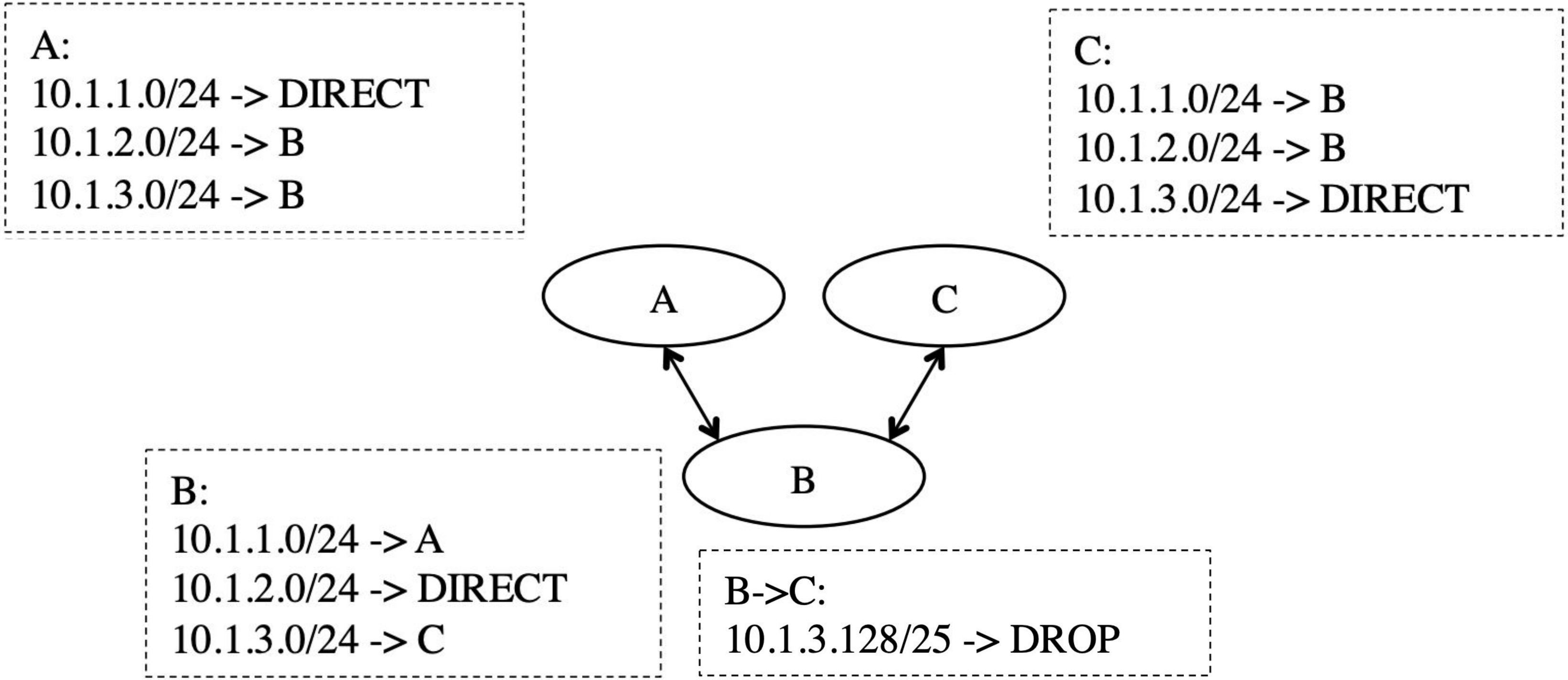
A:
10.1.1.0/24 -> DIRECT
10.1.2.0/24 -> B
10.1.3.0/24 -> B



Example - Anteater (SIGCOMM'11)



Example - Anteater (SIGCOMM'11)



Example - Anteater (SIGCOMM'11)

A:
10.1.1.0/24 -> DIRECT
10.1.2.0/24 -> B
10.1.3.0/24 -> B

B:
10.1.1.0/24 -> A
10.1.2.0/24 -> DIRECT
10.1.3.0/24 -> C

B->C:
10.1.3.128/25 -> DROP

C:
10.1.1.0/24 -> B
10.1.2.0/24 -> B
10.1.3.0/24 -> DIRECT

Example - Anteater (SIGCOMM'11)

A:
10.1.1.0/24 -> DIRECT
10.1.2.0/24 -> B
10.1.3.0/24 -> B

B:
10.1.1.0/24 -> A
10.1.2.0/24 -> DIRECT
10.1.3.0/24 -> C

C:
10.1.1.0/24 -> B
10.1.2.0/24 -> B
10.1.3.0/24 -> DIRECT

B->C:
10.1.3.128/25 -> DROP

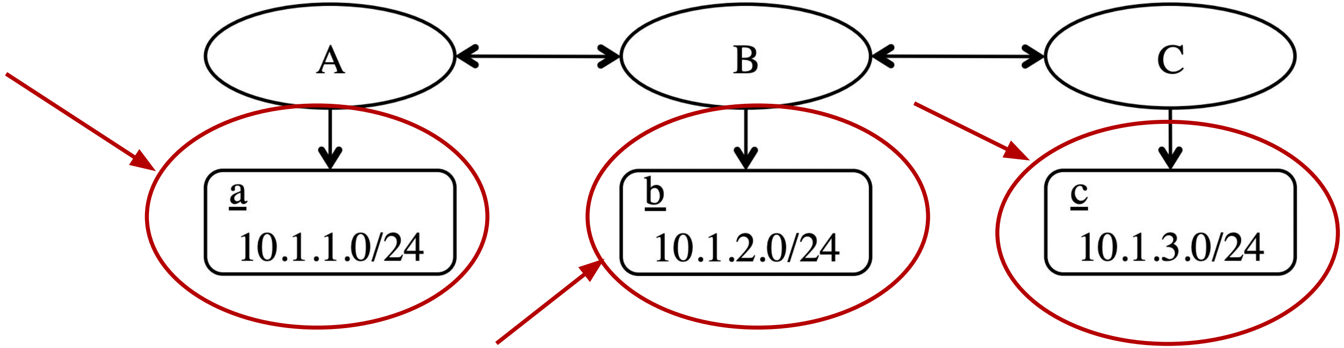
Example - Anteater (SIGCOMM'11)

A:
10.1.1.0/24 -> DIRECT
10.1.2.0/24 -> B
10.1.3.0/24 -> B

B:
10.1.1.0/24 -> A
10.1.2.0/24 -> DIRECT
10.1.3.0/24 -> C

C:
10.1.1.0/24 -> B
10.1.2.0/24 -> B
10.1.3.0/24 -> DIRECT

B->C:
10.1.3.128/25 -> DROP



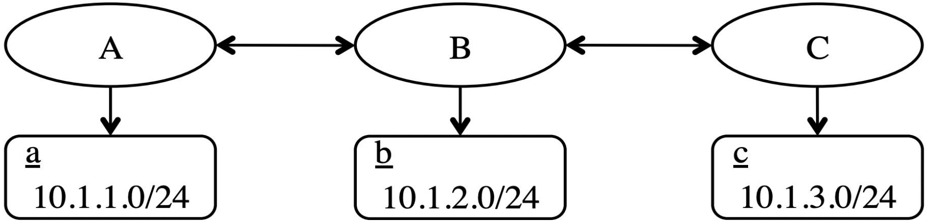
Example - Anteater (SIGCOMM'11)

A:
10.1.1.0/24 -> DIRECT
10.1.2.0/24 -> B
10.1.3.0/24 -> B

B:
10.1.1.0/24 -> A
10.1.2.0/24 -> DIRECT
10.1.3.0/24 -> C

C:
10.1.1.0/24 -> B
10.1.2.0/24 -> B
10.1.3.0/24 -> DIRECT

B->C:
10.1.3.128/25 -> DROP



Model each bit in the packet as a boolean variable.

- The rules only use destination IP, so we only model the 32 bits in the destination IP address.

`P(x, y)`: boolean formula describing which packets can go from **`x`** to **`y`**.

Example - Anteater (SIGCOMM'11)

A:
10.1.1.0/24 -> DIRECT
10.1.2.0/24 -> B
10.1.3.0/24 -> B

B:
10.1.1.0/24 -> A
10.1.2.0/24 -> DIRECT
10.1.3.0/24 -> C

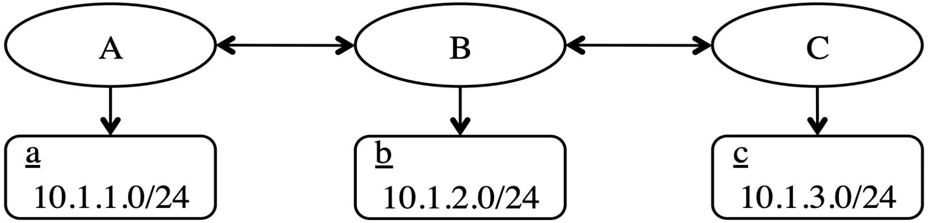
C:
10.1.1.0/24 -> B
10.1.2.0/24 -> B
10.1.3.0/24 -> DIRECT

B->C:
10.1.3.128/25 -> DROP

P(x, y): boolean formula describing which packets can go from **x** to **y**.

$$P(A, a) = \texttt{dst ip} =_{24} 10.1.1.0$$

$$P(A, B) = \texttt{dst ip} =_{24} 10.1.2.0 \lor \texttt{dst ip} =_{24} 10.1.3.0$$



Example - Anteater (SIGCOMM'11)

A:
10.1.1.0/24 -> DIRECT
10.1.2.0/24 -> B
10.1.3.0/24 -> B

B:
10.1.1.0/24 -> A
10.1.2.0/24 -> DIRECT
10.1.3.0/24 -> C

C:
10.1.1.0/24 -> B
10.1.2.0/24 -> B
10.1.3.0/24 -> DIRECT

B->C:
10.1.3.128/25 -> DROP

P(x, y): boolean formula describing which packets can go from **x** to **y**.

$\mathtt{P(A,\ a)} = \mathtt{dst\ ip} =_{24} \mathtt{10.1.1.0}$

$\mathtt{P(A,\ B)} = \mathtt{dst\ ip} =_{24} \mathtt{10.1.2.0}$
$\qquad \vee\ \mathtt{dst\ ip} =_{24} \mathtt{10.1.3.0}$

$\mathtt{dst\ ip} =_{w} \mathtt{prefix}$

is a shorthand for

$\bigwedge_{32-w \le i \le 32} (\mathtt{dst\ ip[i]} = \mathtt{prefix[i]})$

Example - Anteater (SIGCOMM'11)

A:
10.1.1.0/24 -> DIRECT
10.1.2.0/24 -> B
10.1.3.0/24 -> B

B:
10.1.1.0/24 -> A
10.1.2.0/24 -> DIRECT
10.1.3.0/24 -> C

C:
10.1.1.0/24 -> B
10.1.2.0/24 -> B
10.1.3.0/24 -> DIRECT

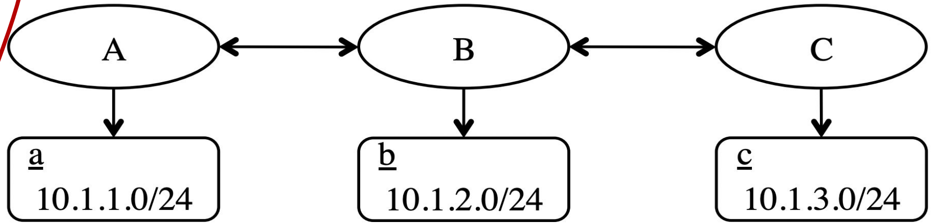
B->C:
10.1.3.128/25 -> DROP

P(x, y): boolean formula describing which packets can go from **x** to **y**.

$P(B,A) = \text{dst ip} =_{24} 10.1.1.0$

$P(B, b) = \text{dst ip} =_{24} 10.1.2.0$

$P(B, C) = \text{dst ip} =_{24} 10.1.3.0$
$\wedge\ \text{dst ip} \neq_{25} 10.1.3.128$



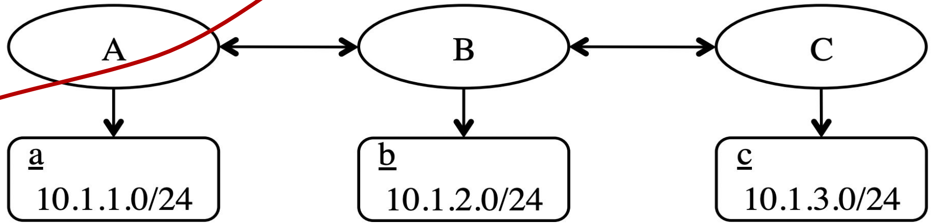
Example - Anteater (SIGCOMM'11)

A:
10.1.1.0/24 -> DIRECT
10.1.2.0/24 -> B
10.1.3.0/24 -> B

B:
10.1.1.0/24 -> A
10.1.2.0/24 -> DIRECT
10.1.3.0/24 -> C

C:
10.1.1.0/24 -> B
10.1.2.0/24 -> B
10.1.3.0/24 -> DIRECT

B->C:
10.1.3.128/25 -> DROP



`P(x, y)`: boolean formula describing which packets can go from **`x`** to **`y`**.

$$\texttt{P(C, B)} = \texttt{dst ip} =_{24} \texttt{10.1.1.0} \lor \texttt{dst ip} =_{24} \texttt{10.1.2.0}$$

$$\texttt{P(C, c)} = \texttt{dst ip} =_{24} \texttt{10.1.3.0}$$

Example - Anteater (SIGCOMM'11)

- Can A reach C?
- Anteater uses a simple graph algorithm to construct the boolean formula that describe all the packets that can reach C from A using P(x, y)
- That formula is **`P(A, B)`** ∧ **`P(B, C)`**
- The formula is given to a SAT solver to check if any assignment to the boolean variables, i.e., any destination IP address, exists that can go from A to C
- If no, no packets can reach C from A

Example - Anteater (SIGCOMM'11)

- This was just a simple example
- Anteater shows how to use a similar approach to check for absence of loops and black holes, among other properties.

Reasoning about network forwarding behavior

- Anteater models network behavior as SAT formulas and uses a SAT solver for their analysis.
- Since then, there has been several other proposals for other ways for both modeling and analysis

Reasoning about network forwarding behavior

- Since then, there has been several other proposals for other ways for both modeling and analysis
- Header Space Analysis (HSA) (NSDI'12)
  - models sets of K-bit packets as subspaces in a K-dimensional space
  - uses set operations for analysis
- Veriflow (NSDI'13)
  - uses a trie to find equivalence classes (ECs) of packets
  - models the forwarding behavior of ECs using a forwarding graph
  - analyzes the network behavior using graph algorithms
- There has been a lot more! (see netverify.fun for a survey)

Formal methods in networking

- Data-plane verification
  - Model and analyze the forwarding rules on the data plane
  - Anteater, HSA, Veriflow, …
- Control-plane verification
  - Model and analyze the control-plane protocols that configure the data plane
- Stateful and programmable data planes

Formal methods in networking

- Analyzing DNS
  - Is there a query under our domain that is sent for resolution to a name server, not under our domain?
- Analyzing performance
  - Is there an input traffic pattern under which the network provides high latency?

Formal methods in networking industry

- Large cloud providers are integrating formal methods into their network operations
  - Microsoft, Amazon, Google, Alibaba, …
  - "Be sure before shipping – the need for safety in clouds" - Dave Maltz keynote in the netverify'21 workshop organized by Microsoft and Google
- Several startup companies
  - Forward Networks, Veriflow, Intentionet, …

How does this all relate to programmable networks?

- Automated testing and verification did not start with and is not limited to programmable networks.
- But, programming abstractions for a single device or collection of devices provides extra opportunities.
  - We can reuse so much of the existing knowledge, expertise, and tools for program verification in the formal methods and PL community
  - In our "network" programs, we already have accurate well-defined specifications of network functionality.
  - We can verify the compilers (or their output) to provide end-to-end verified tool chains
  - …

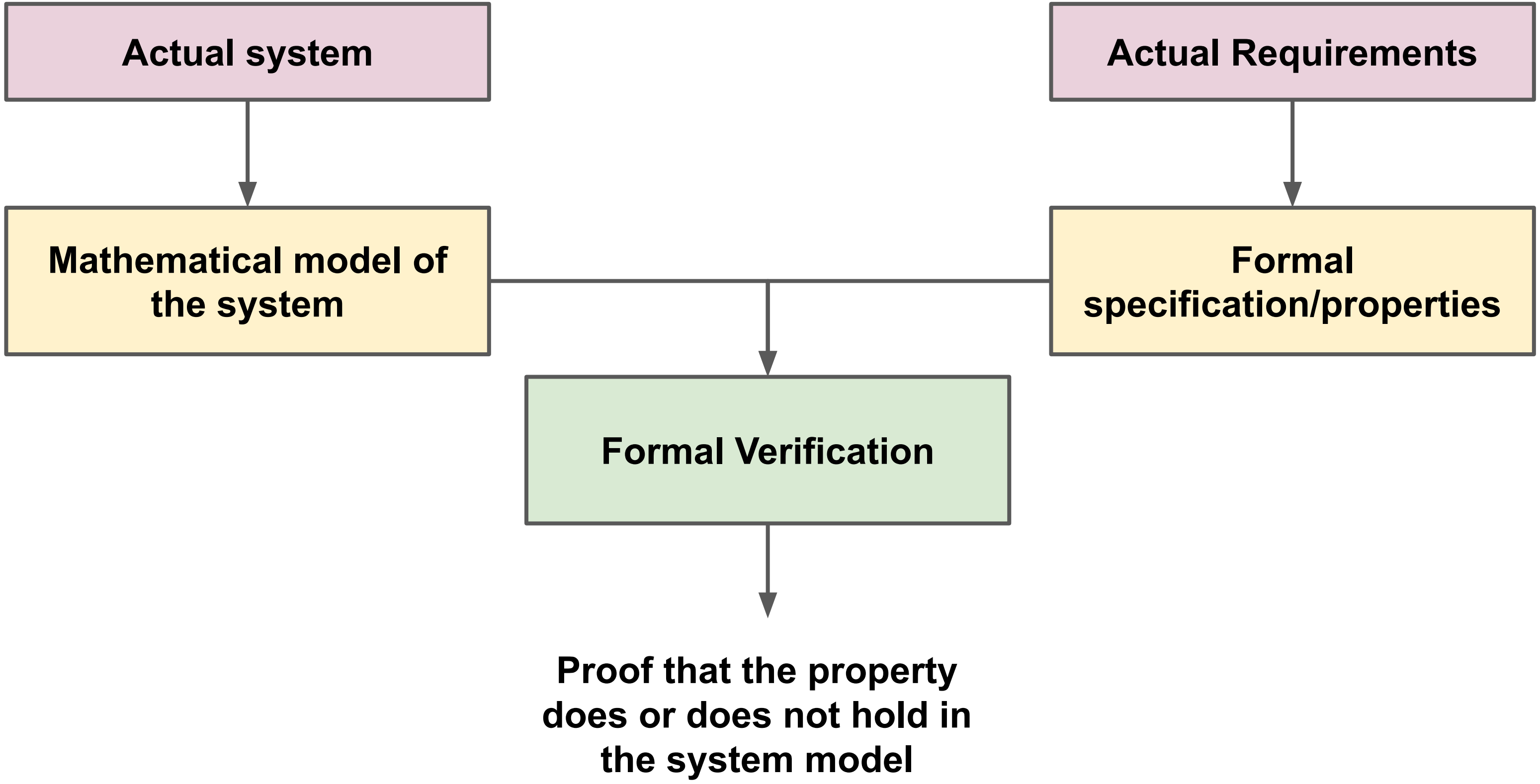
What's next?

- So far, we have convinced ourselves that using formal methods in networking is both essential and possible
- Now, we need to make it usable in a more widespread manner in real-world networks.
- What is missing?

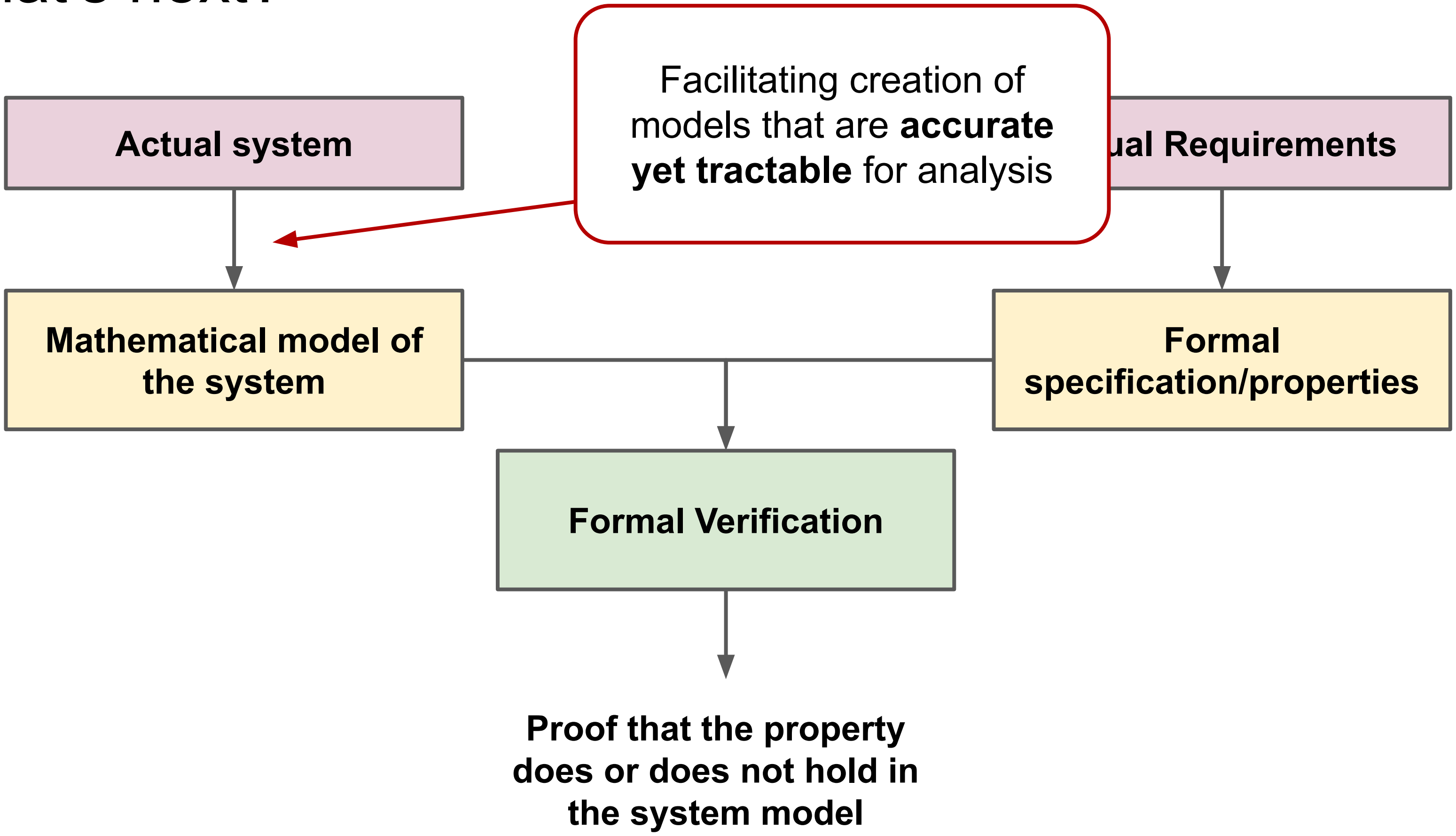
What's next?

- Scale
  - Formal methods tools don't scale well :)
  - There is evidence that they can scale to large network for certain networks and certain properties with lots of optimizations
  - One way forward is "modular" verification, where we verify smaller subsets of the network independently and then combine the results.
  - So, there is hope but also still a long way to go
- Functionalities and properties beyond forwarding
  - network functions, network performance, …

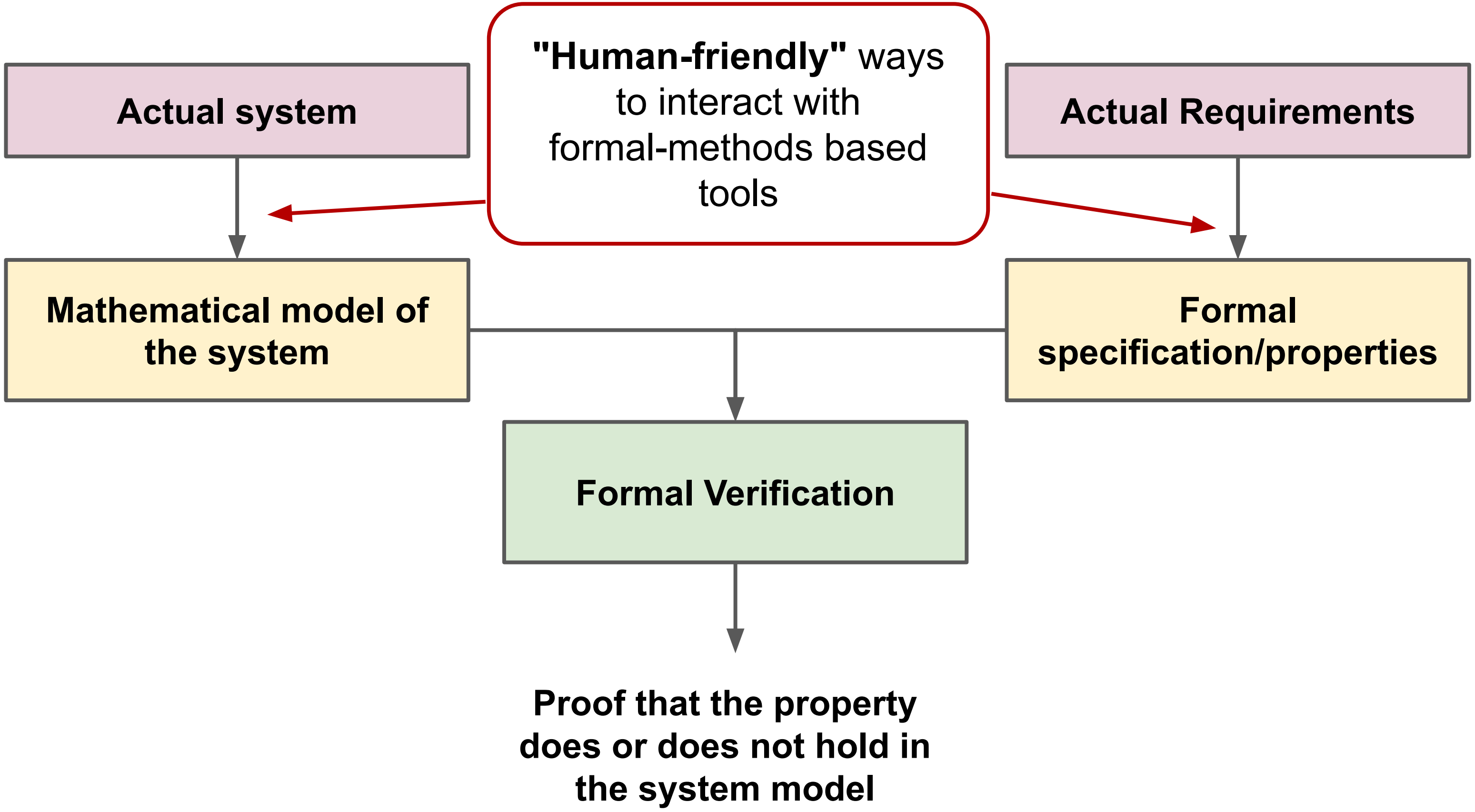
What's next?


Actual system
Actual Requirements
Mathematical model of the system
Formal specification/properties
Formal Verification
Proof that the property does or does not hold in the system model

What's next?


Facilitating creation of models that are **accurate yet tractable** for analysis
Actual system
ual Requirements
Mathematical model of the system
Formal specification/properties
Formal Verification
Proof that the property does or does not hold in the system model

What's next?



Actual system
"Human-friendly" ways to interact with formal-methods based tools
Actual Requirements
Mathematical model of the system
Formal specification/properties
Formal Verification
Proof that the property does or does not hold in the system model

Paper 1: p4v: Practical Verification for Programmable Data Planes

- A tool for verifying properties about P4 programs
  - General safety properties, e.g., avoiding read/writes to invalid headers
  - Program-specific properties specified using assert statements
- Has to work around the fact that the some data-plane rules come from the control plane and are only known at run-time

Paper 2: Validating Datacenters At Scale

- Describes the tools used in Microsoft Azure's network for verifying ACLs and forwarding rules
- To scale, they use domain-specific insights to simplify the analysis
  - Structural properties of the topology
  - Decompose what they want to validate into checks on local devices
  - …

Additional Resources

- netverify.fun
  - History and survey of verification tools
  - Articles from experts about what's new in the area
- Network verification and synthesis course from University of Washington
- Papers on analyzing DNS and performance, among others