CS6410 Formal Methods for Software Verification

Ashish Mishra

Instructor

- Ashish Mishra
  - Asst. Professor at CSE
  - Before: Postdoc, Purdue CS, Programming Languages Group.
  - Even before: IISc, CSA, PhD
  - Research Goal: Help programmers write correct and trustworthy software.
  - Areas: Programming Languages, Program Verification, Synthesis.

Logistics

- Lecture:
  - When: Slot R (at least for now)
  - Where: CS-LH1
  - Course Website: Please register on Google Classroom link.
  - TBA
- Office Hours:
  - After the class OR with appointment a day before.

Administrative and Logistics

- Lectures 2* (1.5 hrs) a week (Tuesdays and Fridays).
- Academic honesty during the course is essential -
  - for details
- Class attendance and participation has 5% of the marks.
  - The course only makes sense if it is interactive.

Administrative and Logistics

- Pre-requisites
  - Algorithms.
  - Open to other departments after instructor’s consent.

Course Description

- Specification
  - Logics to define the expected behavior of the program.
  - Semantics for the logical statements.
- Verification
  - Methods/Algorithms to automatically check if a program meets its specifications.
- Automated programming.
  - Automating the process of program generation.
- Neurosymbolic approaches to specification, verification and synthesis.
- Practical tools of automated computer-aided reasoning.

Evaluation

- Assignments 15%
- Midterm 20%
- Final Exam 20%
- Surprise quizzes10%
- Class participation/attendance: 5%
- Project 30%

Resources

- Course Textbook:
  - [BM] The Calculus of Computation: Decision Procedures with Applications to Verification. Aaron R Bradley and Zohar Manna
- This will be supplemented with a variety of sources, including:
  - [NN] Nielson, Flemming, Hanne Riis Nielson, and Chris Hankin. Principles of Program Analysis. 1st ed. 1999.
  - [SAT] Biere, Armin. Handbook of Satisfiability. 2009.
  - [MC] Clarke, Edmund M. et al. Handbook of Model Checking. Ed. Edmund M. Clarke et al. 1st ed. 2018.
  - [CC] Cousot, Patrick. Principles of Abstract Interpretation. 2nd ed. 2021.

Now the good stuff… :)

Why should you care?

Programmers make mistakes

Reasoning (About Program Correctness)

Program Verification and Analysis

bugs, bugs everywhere

> A recent study found that 60–70% of vulnerabilities in iOS and macOS are caused by memory unsafety. Microsoft estimates that 70% of all vulnerabilities in their products over the last decade have been caused by memory unsafety. Google estimated that 90% of Android vulnerabilities are memory unsafety. An analysis of 0-days that were discovered being exploited in the wild found that more than 80% of the exploited vulnerabilities were due to memory unsafety.
>
> (Gaynor, 2019)

Why should we care about Correctness?

- When programs have errors, they fail, causing real

Heartbleed: Hundreds of thousands of servers at risk from catastrophic bug

Code error means that websites can leak user details including passwords through 'heartbeat' function used to secure connections

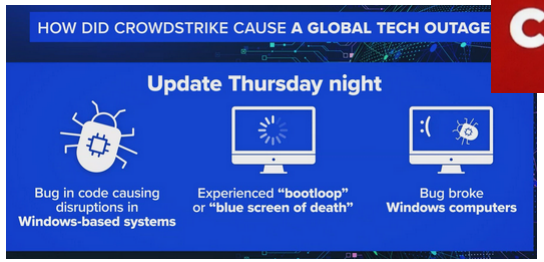
We want our Software to be Correct!

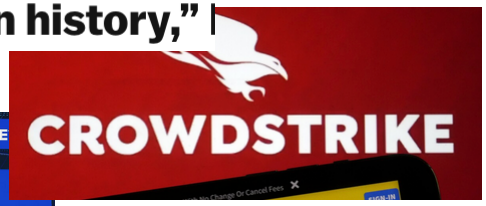
Boeing Fixing New Software Bug on Max; Key Test Flight Nears

- Company says latest flaw shouldn't delay plane's return
- Issue involves aircraft's so-called 'trim' warning system

TECHNOLOGY

The "largest IT outage in history,"





Turing Awards

Dijkstra

Floyd

Hoare

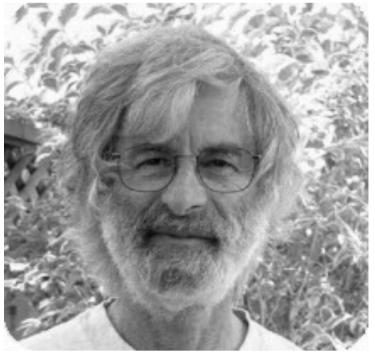
Milner

Pnueli

Clarke

Emerson

Sifakis



Lamport

Success stories

- **Intel** CPU arithmetic and logical operations
- **Microsoft** device drivers
- **Rockwell Collins** AAMP7G microprocessor's partition management
- **Rolls Royce** Trent Series Health Monitoring Units
- **Lockheed Martin** C130J Mission Computers
- **Boeing** "Little Bird" helicopter (seL4 OS-based mission computer)
- **Royal Navy** Ship/Helicopter Operating Limits Unit
- **Airbus 380** primary flight control software
- **Paris Metro** (RATP)
- **NASA** Mars Rover data management subsystem
- **Bombardier ILLBV950L2** railway interlocking system
- **Apple, ARM/SoftBank, Nvidia, IBM, Oracle** RTL
- **AMD** K5 floating point square root microcode
- **Micrium** OS µC/OS-II real-time kernel

SYNOPSYS®

TOYOTA

galois

powered by amazon web services

Astrée

Overview

Software: Traditional vs Formal Methods

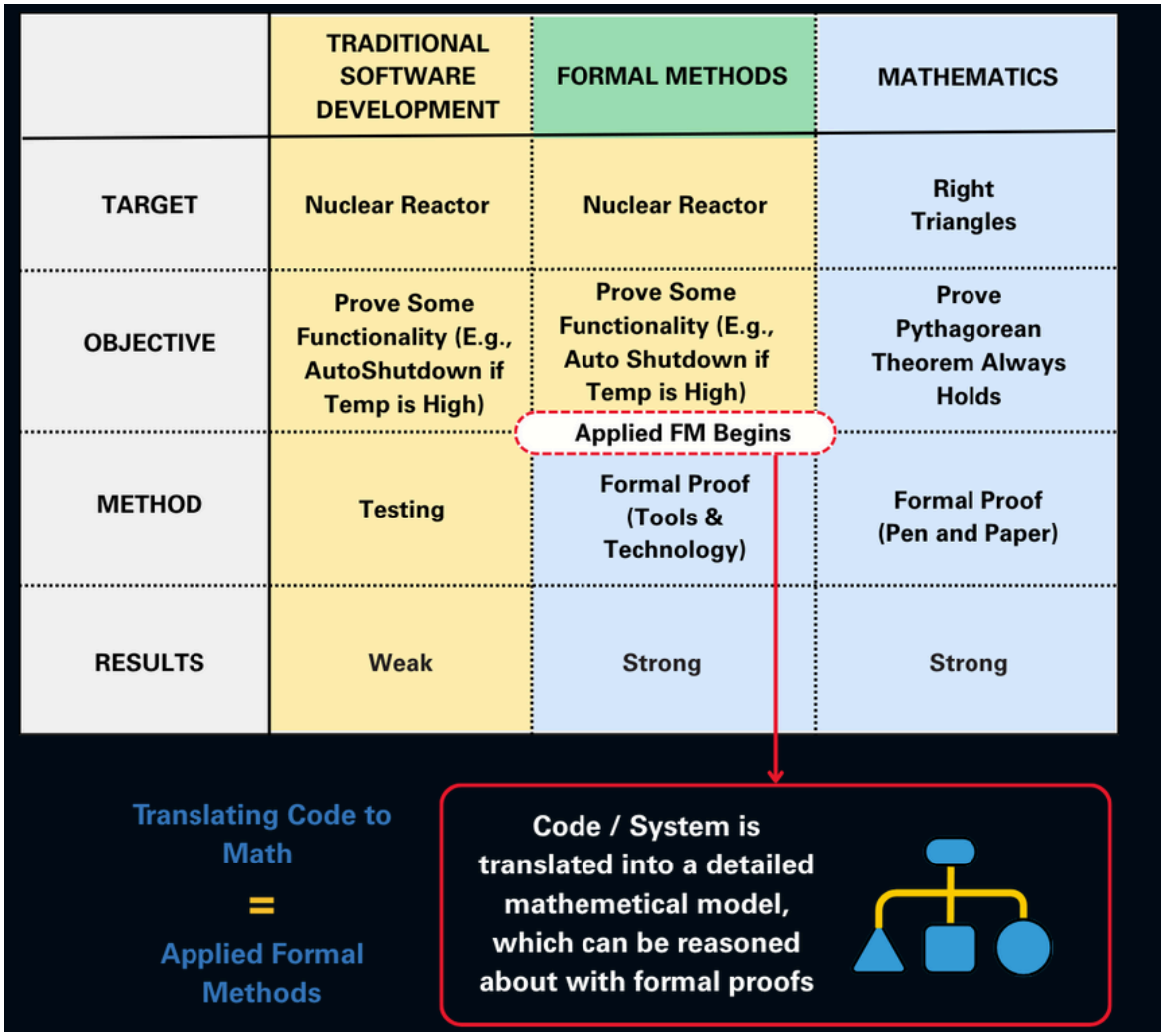


Image: https://www.galois.com/what-are-formal-methods

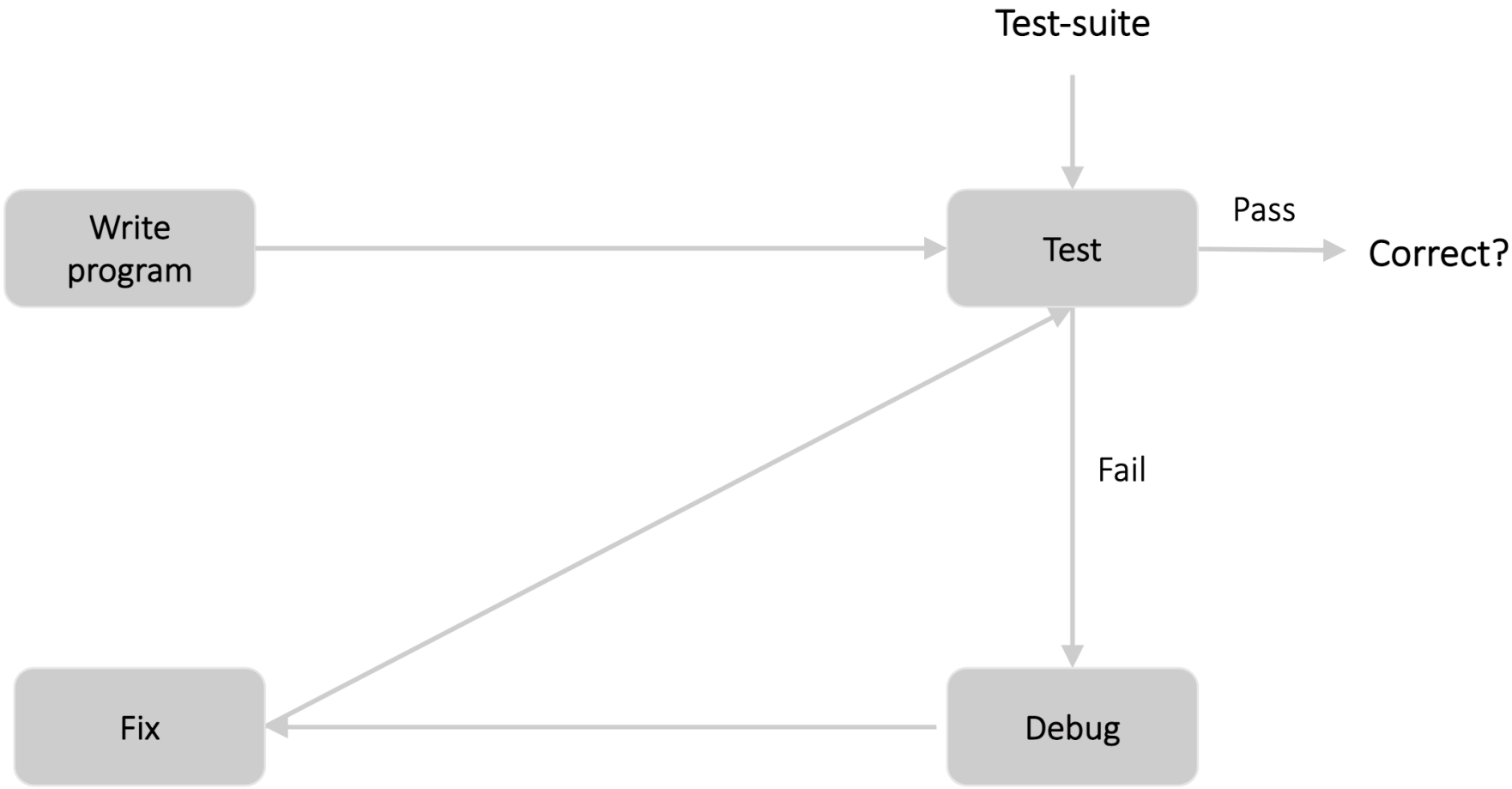
Reactor Control System

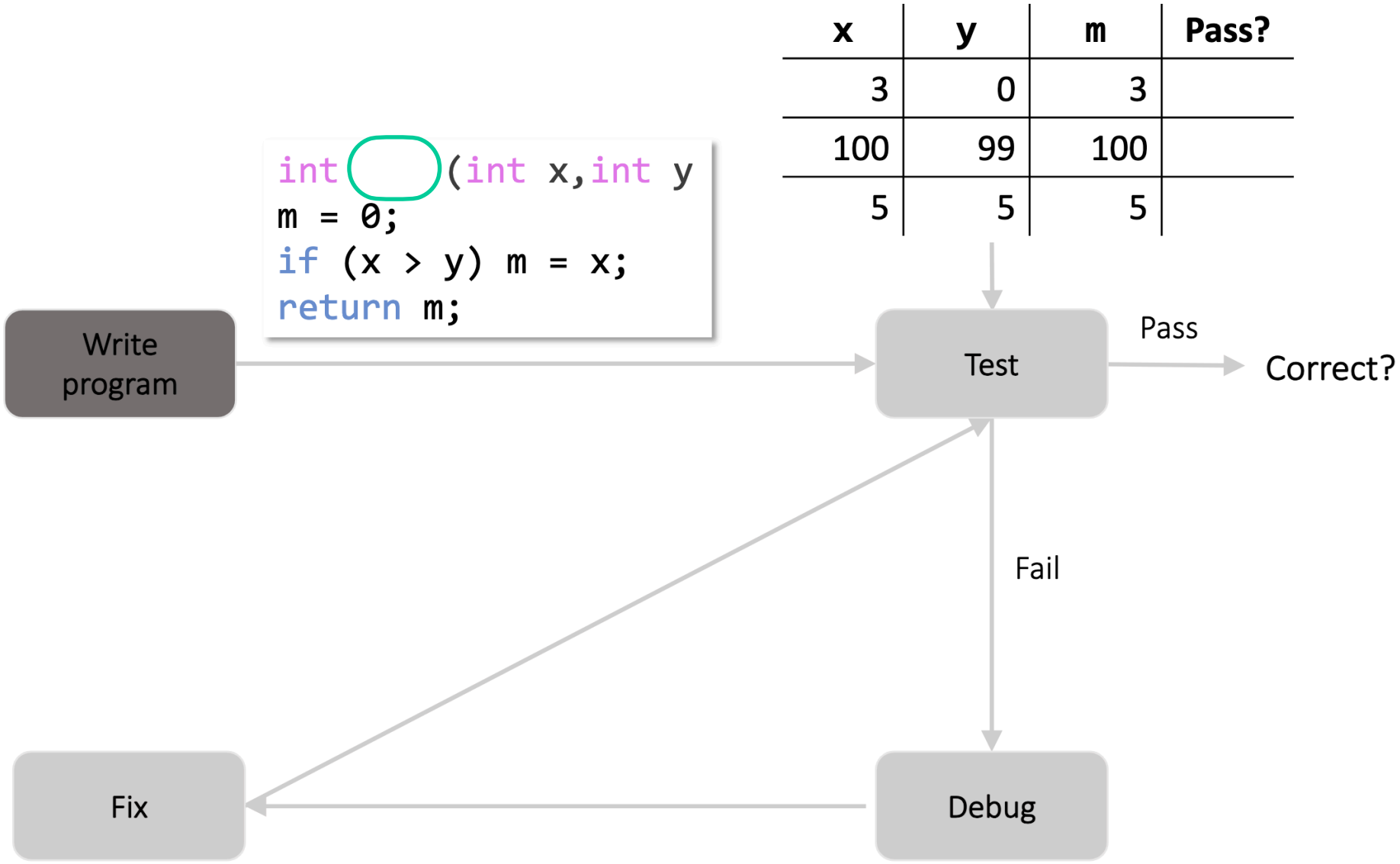
Traditional Development Approach Example:

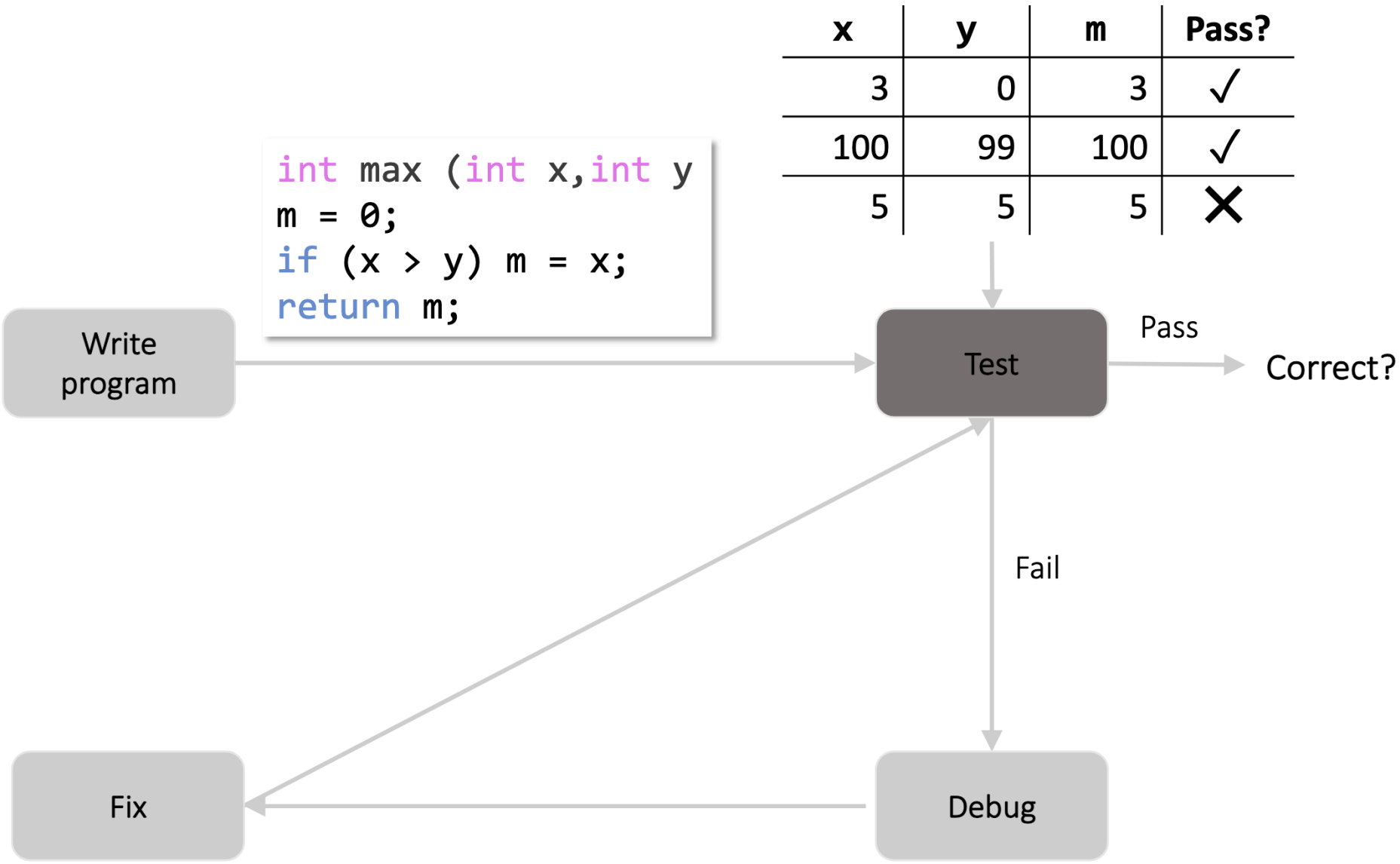
```
bool is_valid_temperature_range(int temperature) {
        return (temperature < 2200);
}
```

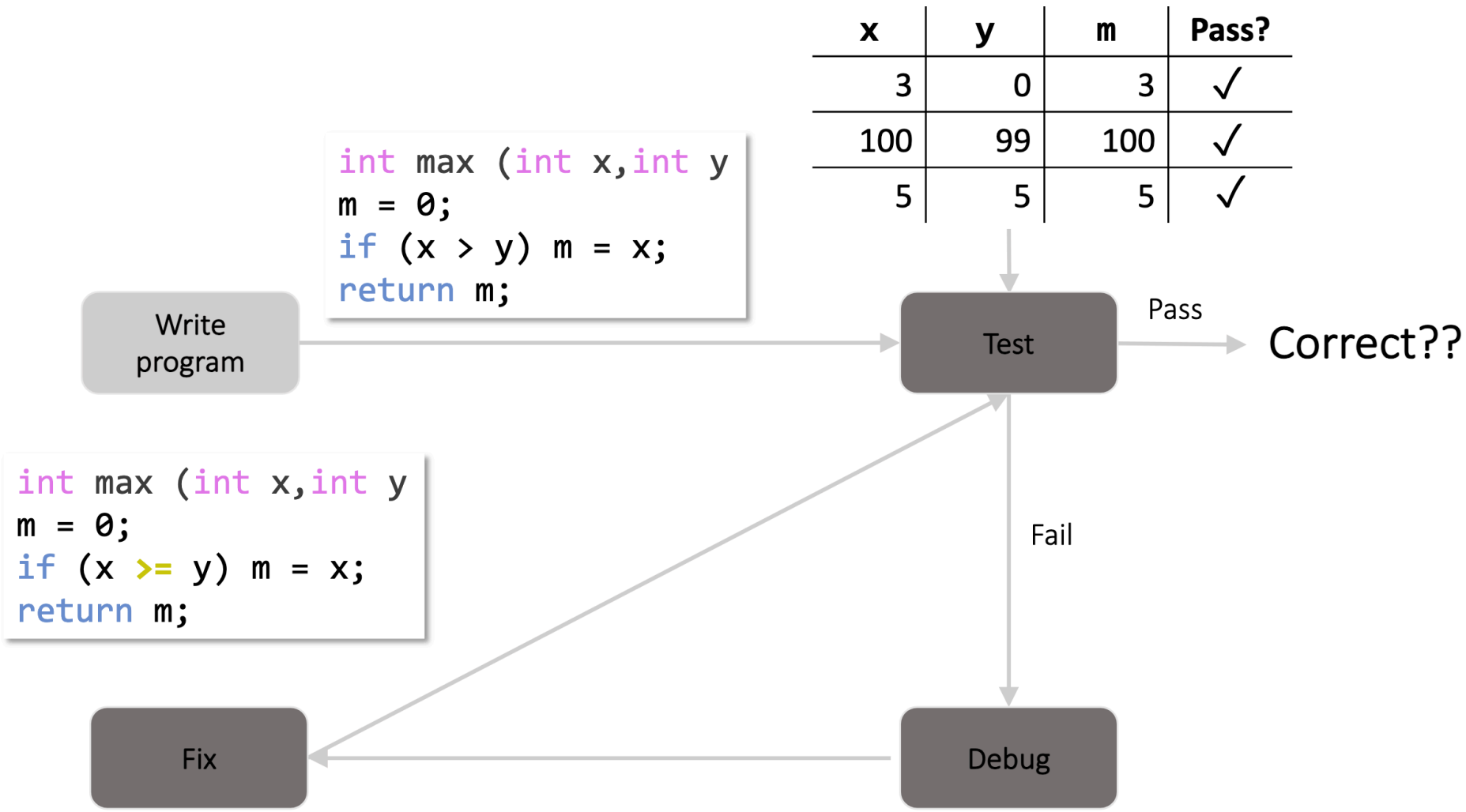
Formal Methods Approach Example:

```
/*
    behavior temperatureInRange:
        assumes (temperature < 2200);
        ensures \result == true;

    behavior temperatureOutOfRange:
        assumes (2200 <= temperature);
        ensures \result == false;
*/

bool is_valid_temperature_range(int temperature) {
    return (temperature < 2200);
}
```


Test-suite
Write program
Test
Pass
Correct?
Fail
Fix
Debug


int (int x,int y
m = 0;
if (x > y) m = x;
return m;
x
y
m
Pass?
3
0
3
100
99
100
5
5
5
Write program
Test
Pass
Correct?
Fail
Fix
Debug

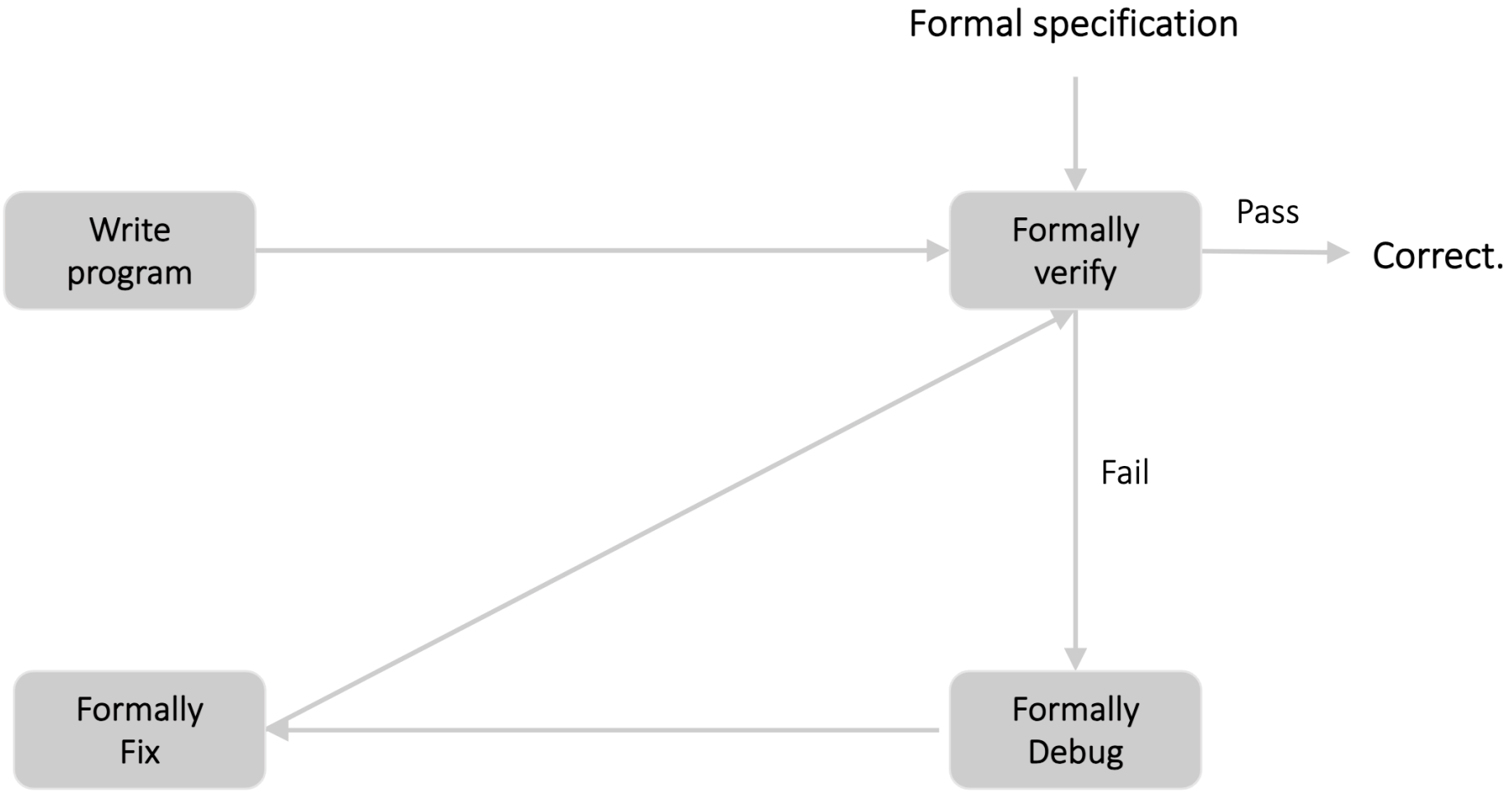

x
y
m
Pass?
3
0
3
✓
100
99
100
✓
5
5
5
✗
int max (int x,int y
m = 0;
if (x > y) m = x;
return m;
Write program
Test
Pass
Correct?
Fail
Fix
Debug

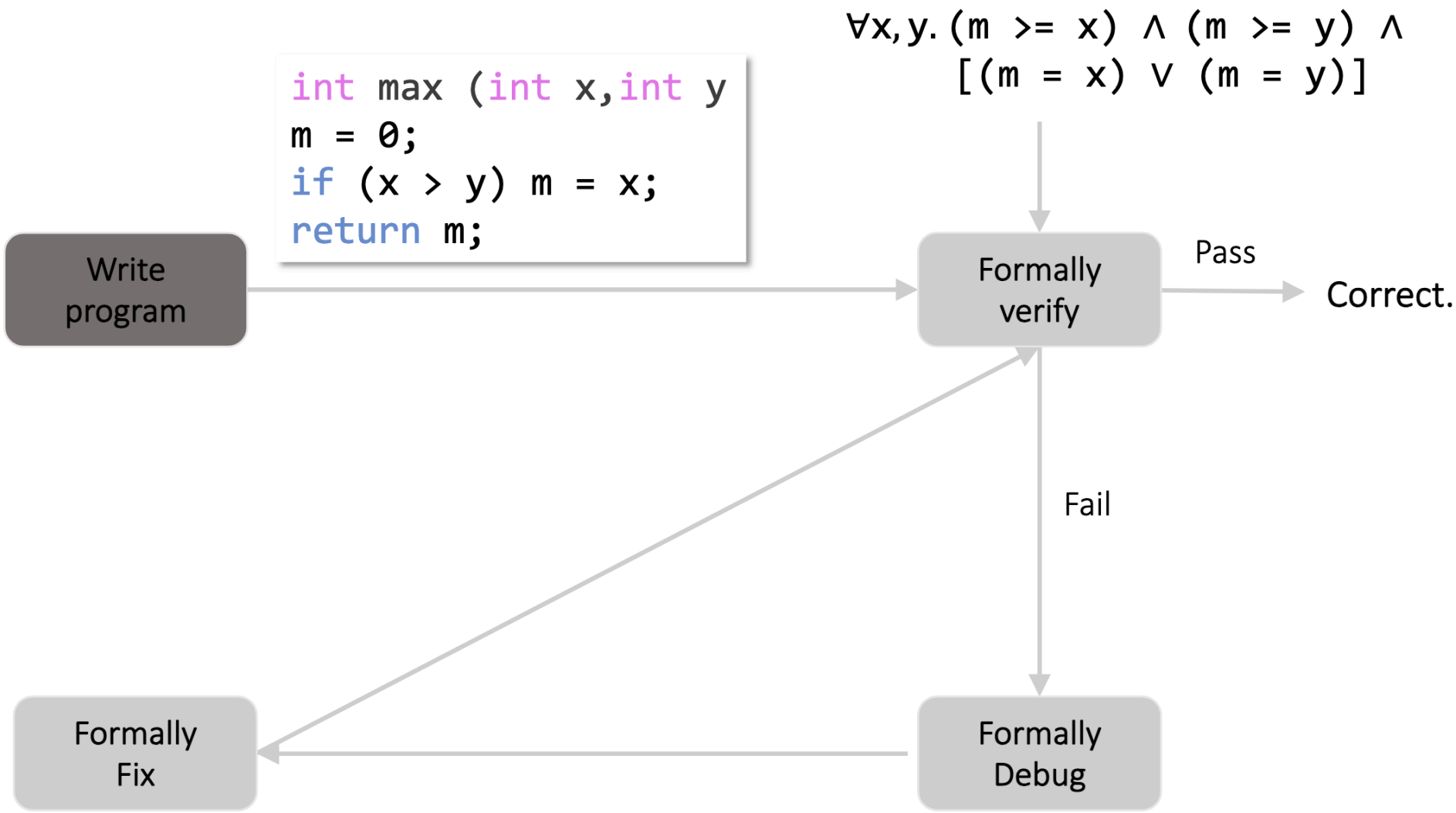

x | y | m | Pass?
3 | 0 | 3 | ✓
100 | 99 | 100 | ✓
5 | 5 | 5 | ✓
int max (int x,int y
m = 0;
if (x > y) m = x;
return m;
Write program
Test
Pass
Correct??
int max (int x,int y
m = 0;
if (x >= y) m = x;
return m;
Fail
Fix
Debug

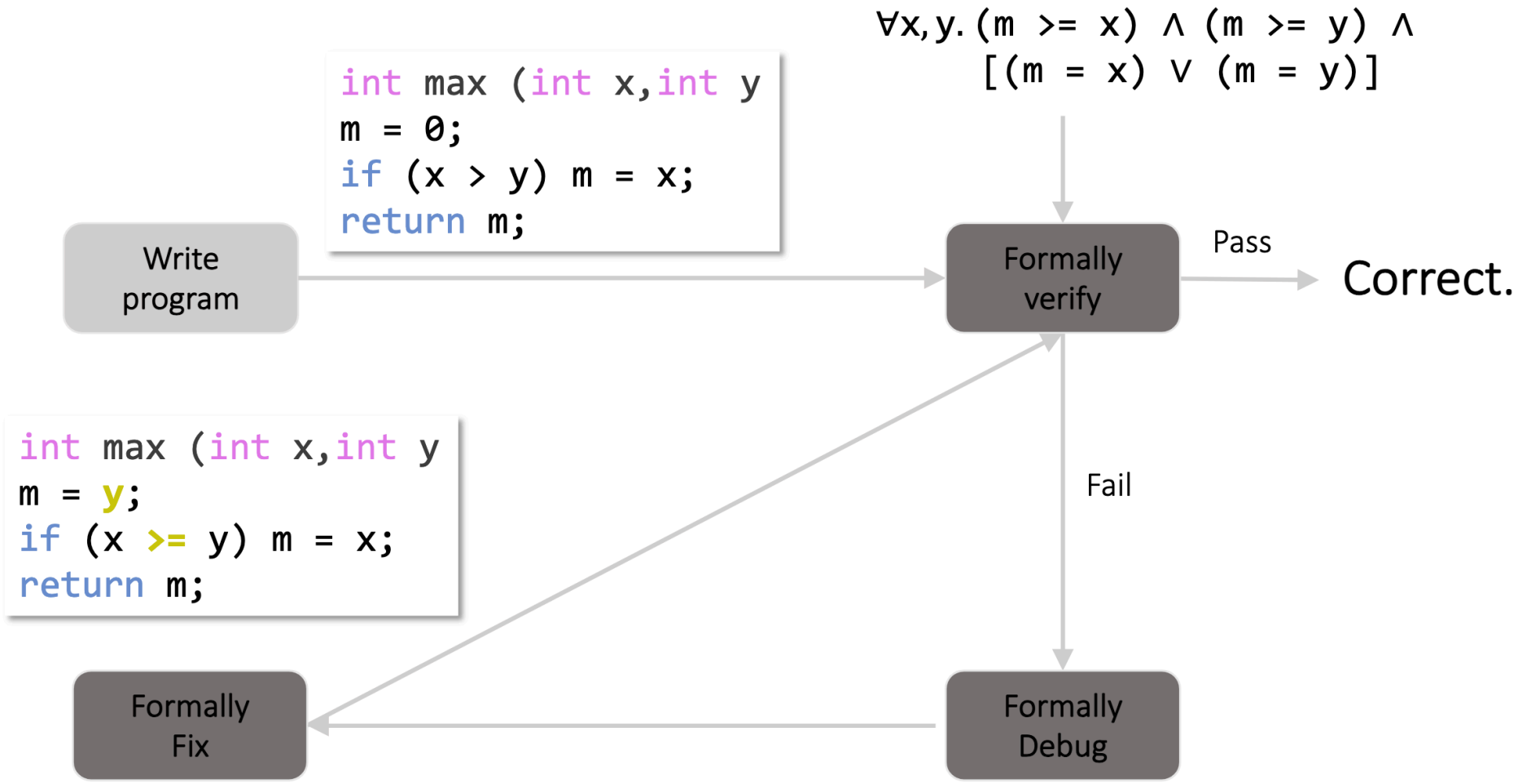
Dijkstra

Testing shows the presence,
not the absence of bugs.

Ergo, testing can fail to show the presence of some bugs!


Formal specification
Write program
Formally verify
Pass
Correct.
Fail
Formally Fix
Formally Debug


∀x, y. (m >= x) ∧ (m >= y) ∧
[(m = x) ∨ (m = y)]
int max (int x,int y
m = 0;
if (x > y) m = x;
return m;
Write program
Formally verify
Pass
Correct.
Fail
Formally Fix
Formally Debug


∀x, y. (m >= x) ∧ (m >= y) ∧
[(m = x) ∨ (m = y)]
int max (int x,int y
m = 0;
if (x > y) m = x;
return m;
Write program
Formally verify
Pass
Correct.
int max (int x,int y
m = y;
if (x >= y) m = x;
return m;
Fail
Formally Fix
Formally Debug

Dijkstra

Testing shows the presence,
not the absence of bugs.

Formal specifications can precisely capture correctness requirements.

Formal verification can **prove** the absence of bugs!

Formal repair can ensure the absence of bugs for programs with bugs!

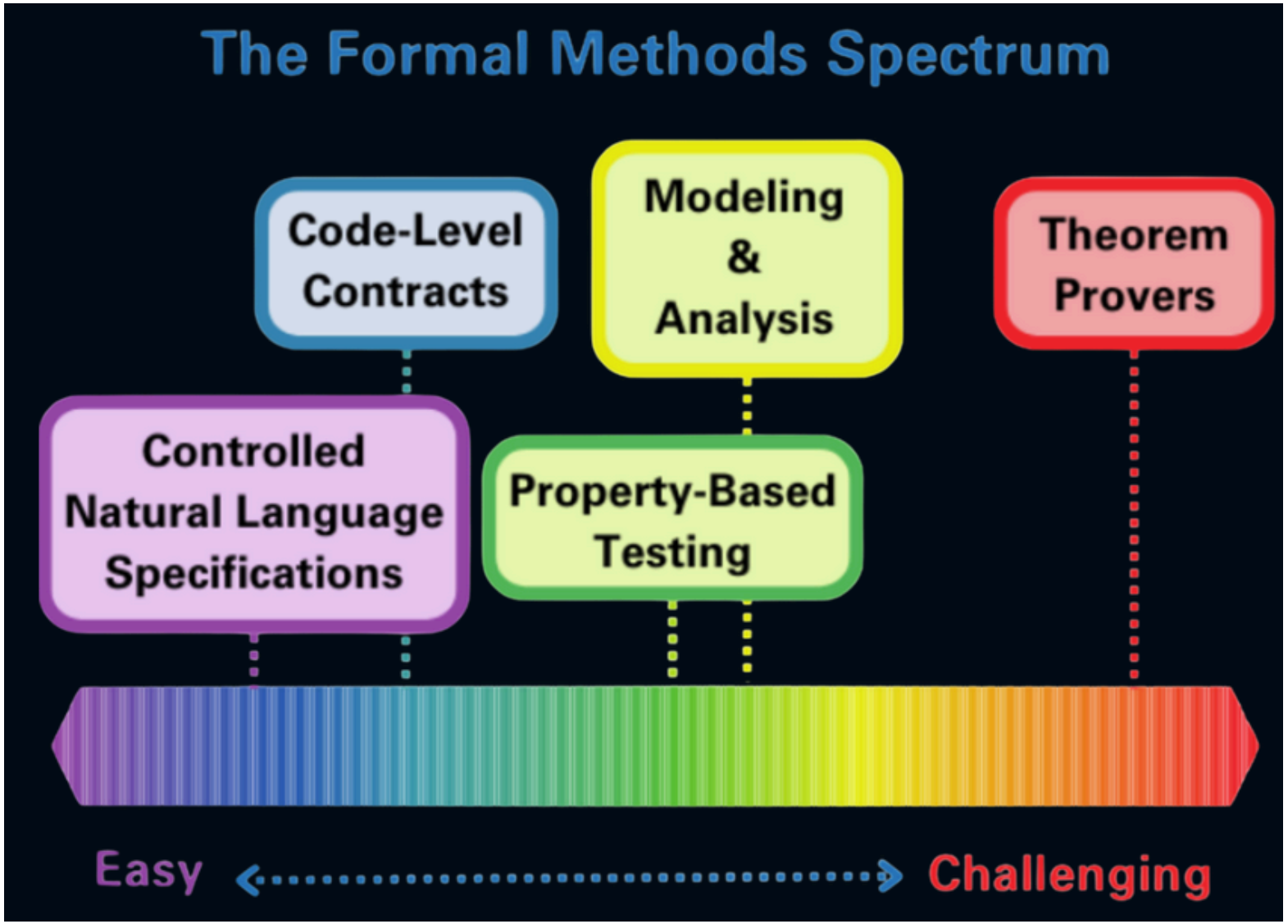
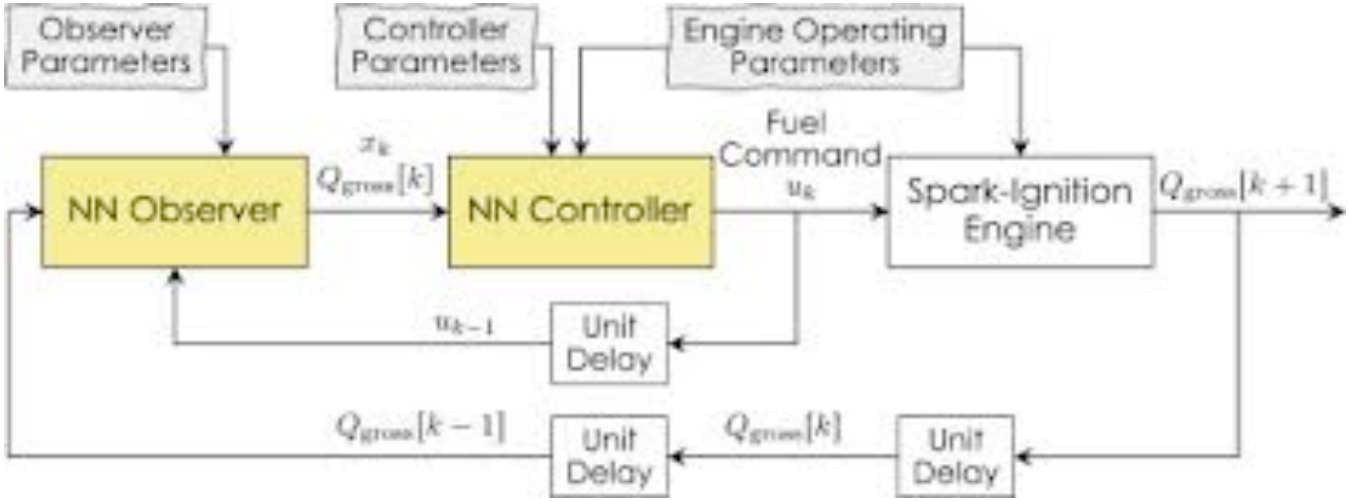


Image: https://www.galois.com/what-are-formal-methods

Formal Methods Examples

- Using automated theorem proving to prove the functional correctness of hash table implementations.
- Creating formal mathematical models to verify implementations of side-channel resistant microchips.
- Using static analysis to ensure memory safety of a C program.
- Using SMT solvers to verify the correctness of a statistical/ML classifier
- Applying formal verification to neural network-based collision avoidance systems that ensure commercial aircraft maintain a safe distance
- Verifying security of blockchain protocols and safety of smart contracts
- Using Rust, which internally uses formal methods to prove memory safety for performance-critical code

An ML Controller: Challenges skyrocket



Basic Techniques still Apply: Spec, Model, Translate, Verify

NN Verification: A few works

2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)

NeuroDiff: Scalable Differential Verification of Neural Networks using Fine-Grained Approximation

Brandon Paulsen
University of Southern California
Los Angeles, California, USA

Jiawei Wang
University of Southern California
Los Angeles, California, USA

Jingbo Wang
University of Southern California
Los Angeles, California, USA

Chao Wang
University of Southern California
Los Angeles, California, USA

Incremental Verification of Neural Networks

SHUBHAM UGARE, University of Illinois Urbana-Champaign, USA
DEBANGSHU BANERJEE, University of Illinois Urbana-Champaign, USA
SASA MISAILOVIC, University of Illinois Urbana-Champaign, USA
GAGANDEEP SINGH, University of Illinois Urbana-Champaign and VMware Research, USA

Complete verification of deep neural networks (DNNs) can exactly determine whether the DNN satisfies a desired trustworthy property (e.g., robustness, fairness) on an infinite set of inputs or not. Despite the tremendous progress to improve the scalability of complete verifiers over the years on individual DNNs, they

ReluDiff: Differential Verification of Deep Neural Networks

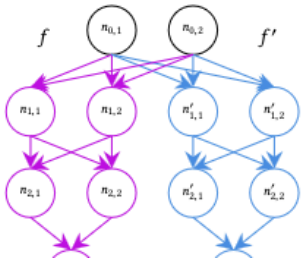
Brandon Paulsen
University of Southern California
Los Angeles, California, USA

Jingbo Wang
University of Southern California
Los Angeles, California, USA

Chao Wang
University of Southern California
Los Angeles, California, USA

ABSTRACT

As deep neural networks are increasingly being deployed in practice, their efficiency has become an important issue. While there are compression techniques for reducing the network's size, energy consumption and computational requirement, they only demonstrate *empirically* that there is no loss of accuracy, but lack formal guarantees of the compressed network, e.g., in the presence of adversarial examples. Existing verification techniques such as Reluplex, ReluVal, and DeepPoly provide formal guarantees, but



Some Basic Principals for Formal Methods Application

Basic Principals

- Precision in Application: What tool/technique to use.
- Specifications are primary and go before implementation.
  - What if you don’t have specs for some legacy code?
    - Learn specifications.
    - Other FM techniques: equivalence testing, binary analysis, some static analyses.
- Validation and verification.
- Abstraction and modeling : Particularly the notion of programs is changing.

Some Examples

Towers to Frequency Matching

Example 2.1. Let $S = \{s_1, \ldots, s_n\}$ be a set of radio stations, each of which has to be allocated one of k transmission frequencies, for some $k < n$. Two stations that are too close to each other cannot have the same frequency. The set of pairs having this constraint is denoted by E. To model this problem, define a set of propositional variables $\{x_{ij} \mid i \in \{1, \ldots, n\}, j \in \{1, \ldots, k\}\}$. Intuitively, variable x_{ij} is set to TRUE if and only if station i is assigned the frequency j. The constraints are:

Constraints at logical formulas

- Every station is assigned at least one frequency:

$$\bigwedge_{i=1}^{n} \bigvee_{j=1}^{k} x_{ij} \ . \tag{2.1}$$

- Every station is assigned not more than one frequency:

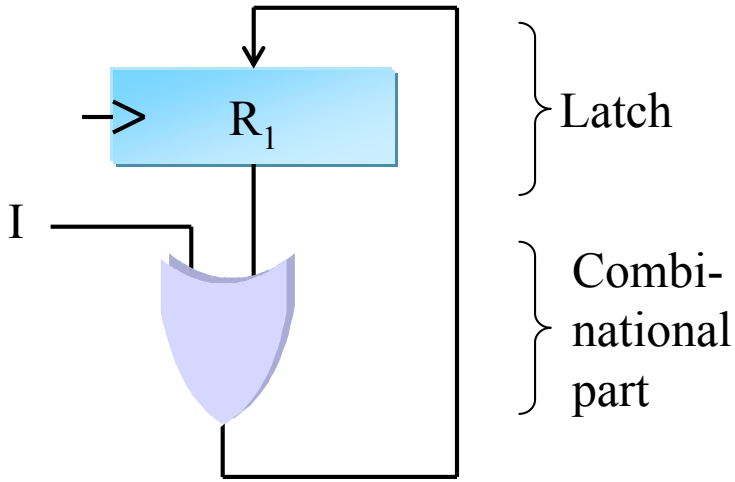
$$\bigwedge_{i=1}^{n} \bigwedge_{j=1}^{k-1} (x_{ij} \implies \bigwedge_{j<t\leq k} \neg x_{it}) \ . \tag{2.2}$$

- Close stations are not assigned the same frequency. For each $(i, j) \in E$,

$$\bigwedge_{t=1}^{k} (x_{it} \implies \neg x_{jt}) \ . \tag{2.3}$$

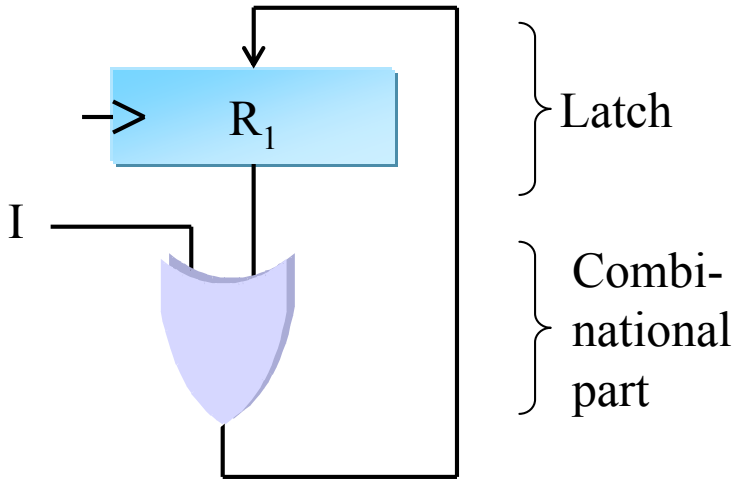
Example: Circuit Transformations

- Circuits consist of combinational gates and latches (registers)



Example: Circuit Transformations

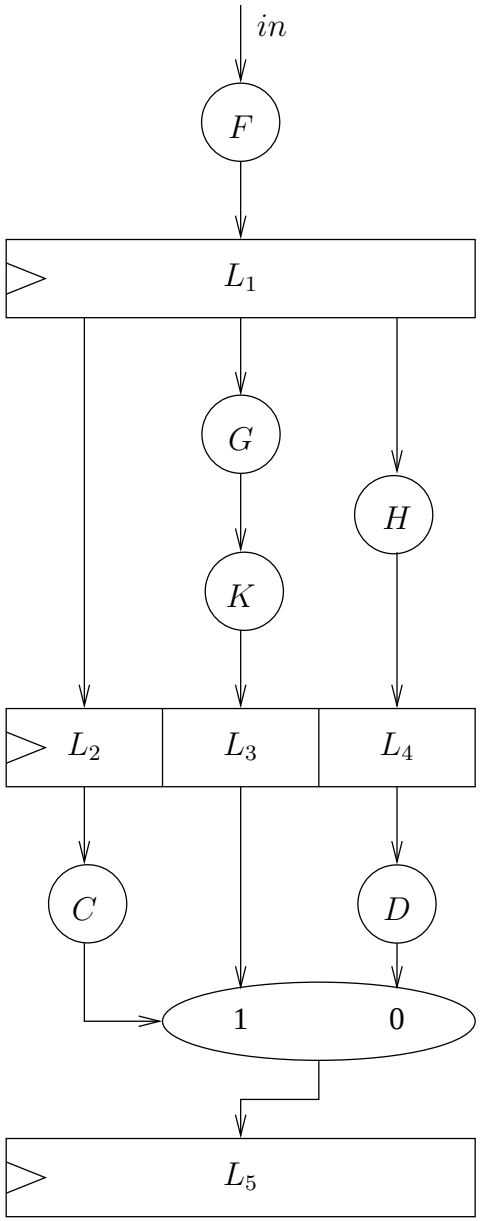
- Circuits consist of combinational gates and latches (registers)



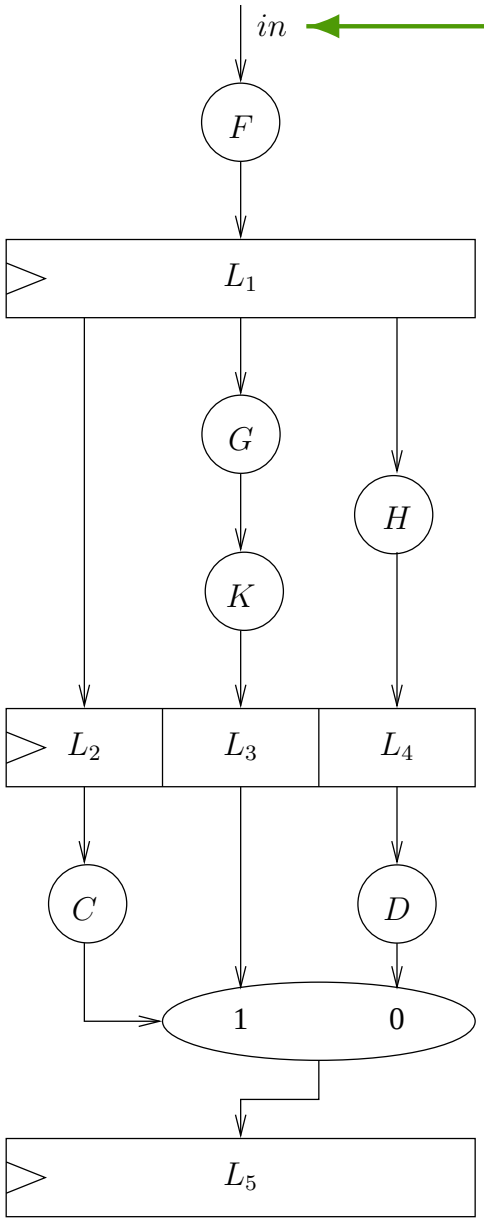
- The combinational gates can be modeled using functions
- The latches can be modeled with variables

$$\begin{aligned} f(x,y) &:= x \vee y \\ R_1' &= f(R_1, I) \end{aligned}$$

Example: Circuit Transformations

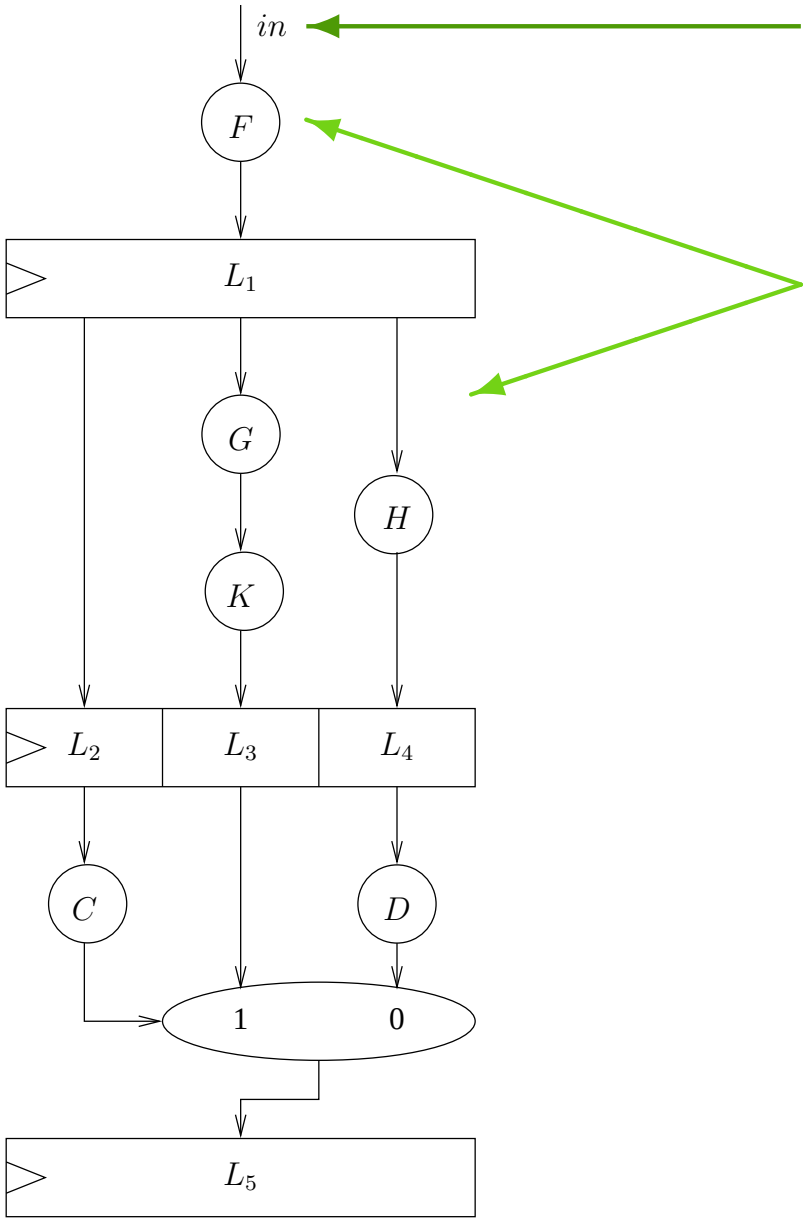


Example: Circuit Transformations



in: a primary input of the circuit

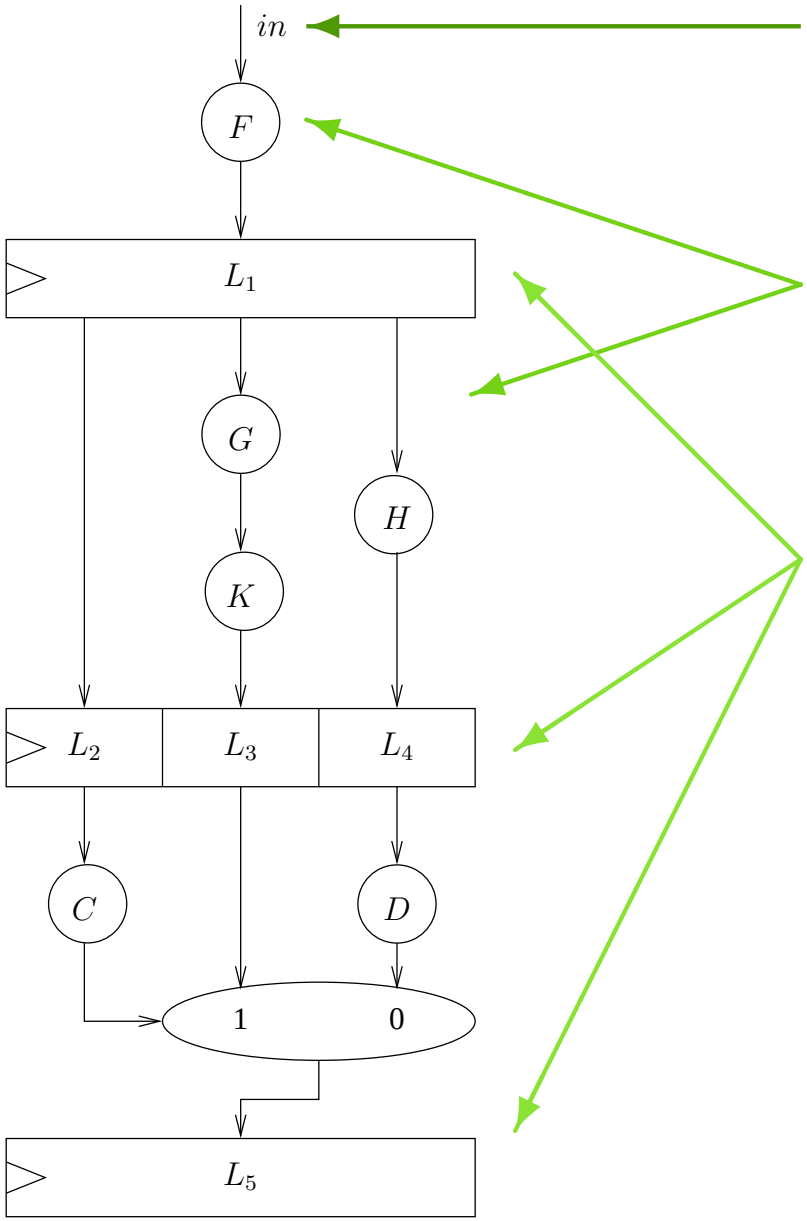
Example: Circuit Transformations



in: a primary input of the circuit

F, G, H, K, D: some functions over bit-vectors

Example: Circuit Transformations

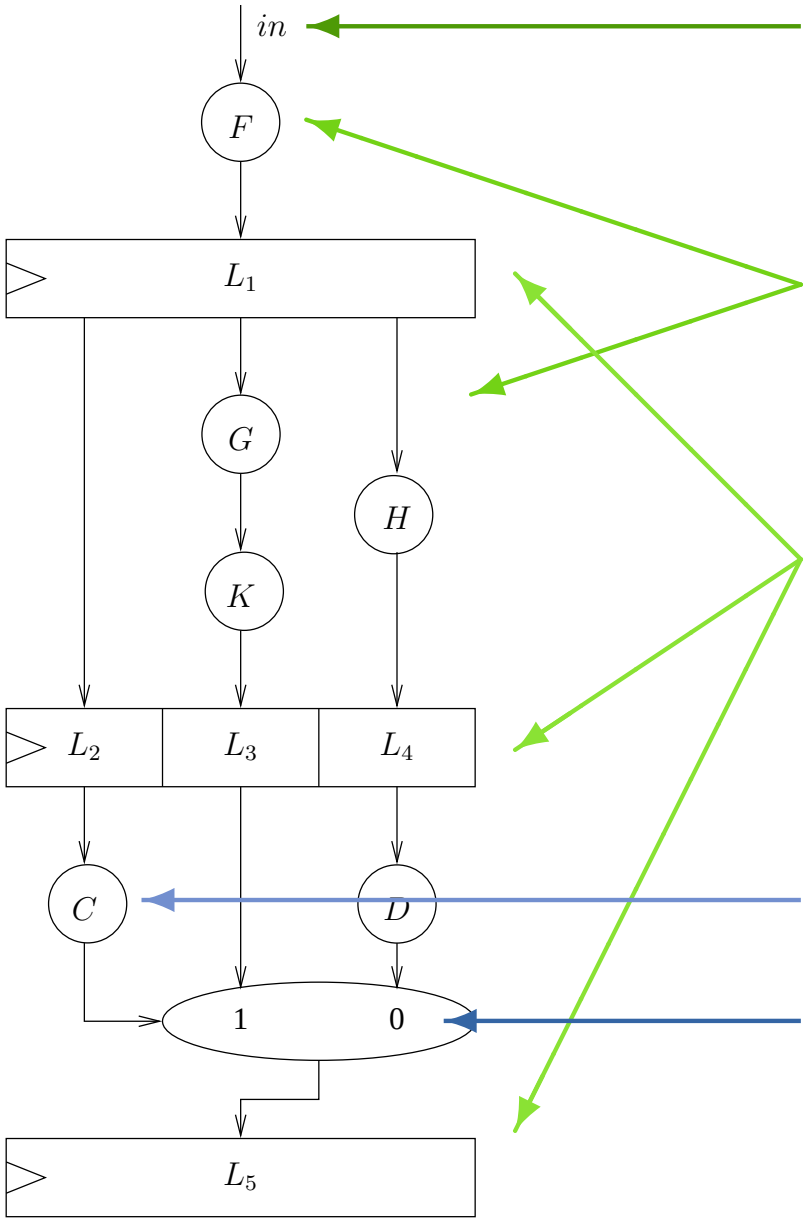


in: a primary input of the circuit

F, G, H, K, D: some functions over bit-vectors

$L_1, \ldots, L_5$: latches (registers)

Example: Circuit Transformations



in: a primary input of the circuit

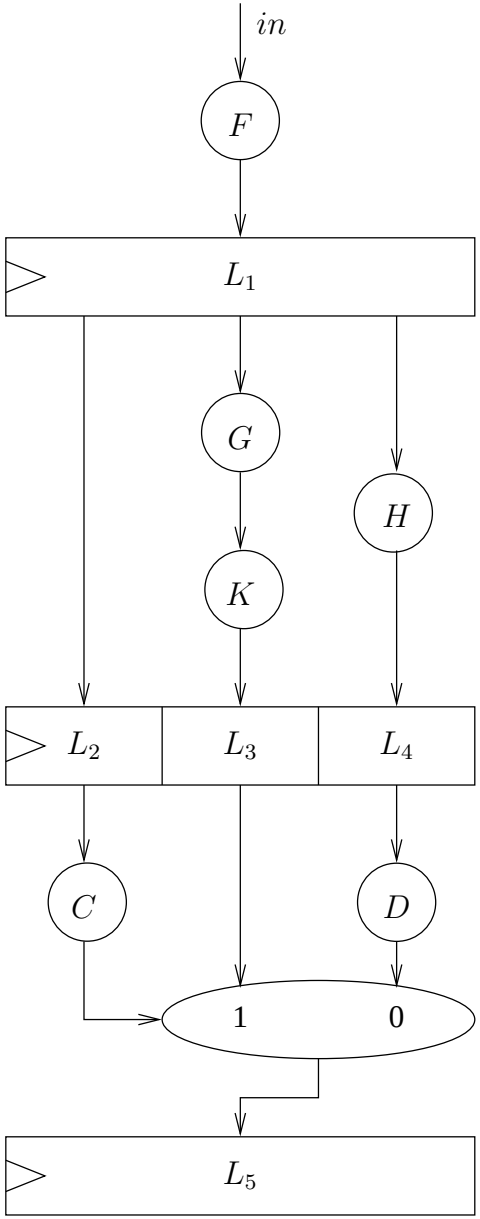
F, G, H, K, D: some functions over bit-vectors

$L_1, \ldots, L_5$: latches (registers)

C: a predicate over bit-vectors

a multiplexer (case-split)

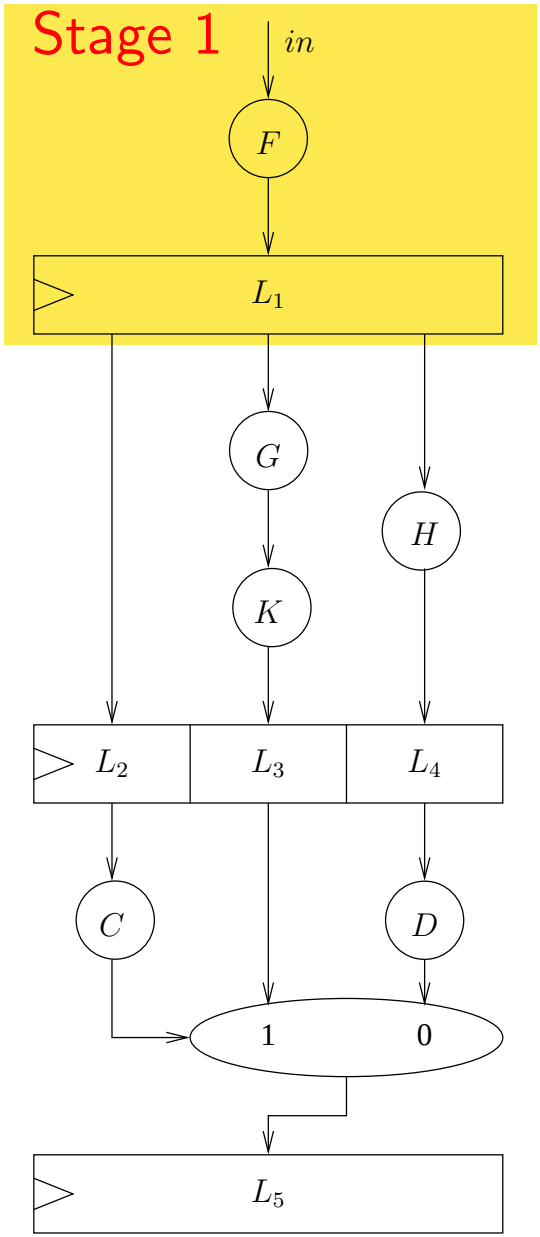
Example: Circuit Transformations



- A pipeline processes data in *stages*
- Data is processed in parallel – as in an assembly line
- Formal model:

$$
\begin{aligned}
L_1 &= f(I) \\
L_2 &= L_1 \\
L_3 &= k(g(L_1)) \\
L_4 &= h(L_1) \\
L_5 &= c(L_2)\,?\,L_3\,:\,l(L_4)
\end{aligned}
$$

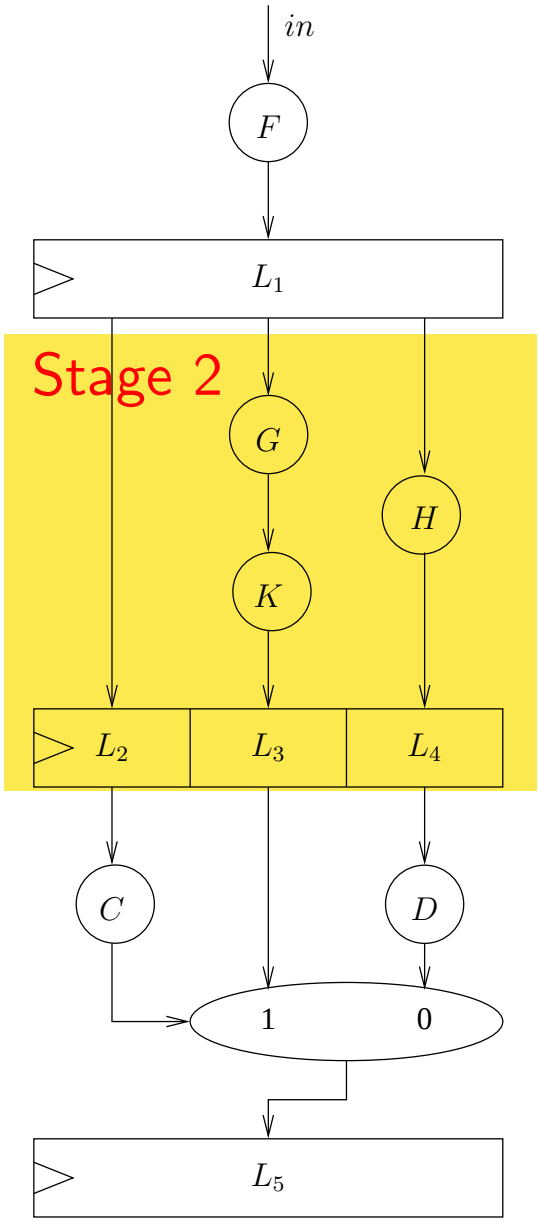
Example: Circuit Transformations



- A pipeline processes data in *stages*
- Data is processed in parallel – as in an assembly line
- Formal model:

$$
\begin{aligned}
L_1 &= f(I) \\
L_2 &= L_1 \\
L_3 &= k(g(L_1)) \\
L_4 &= h(L_1) \\
L_5 &= c(L_2)\,?\,L_3\ :\ l(L_4)
\end{aligned}
$$

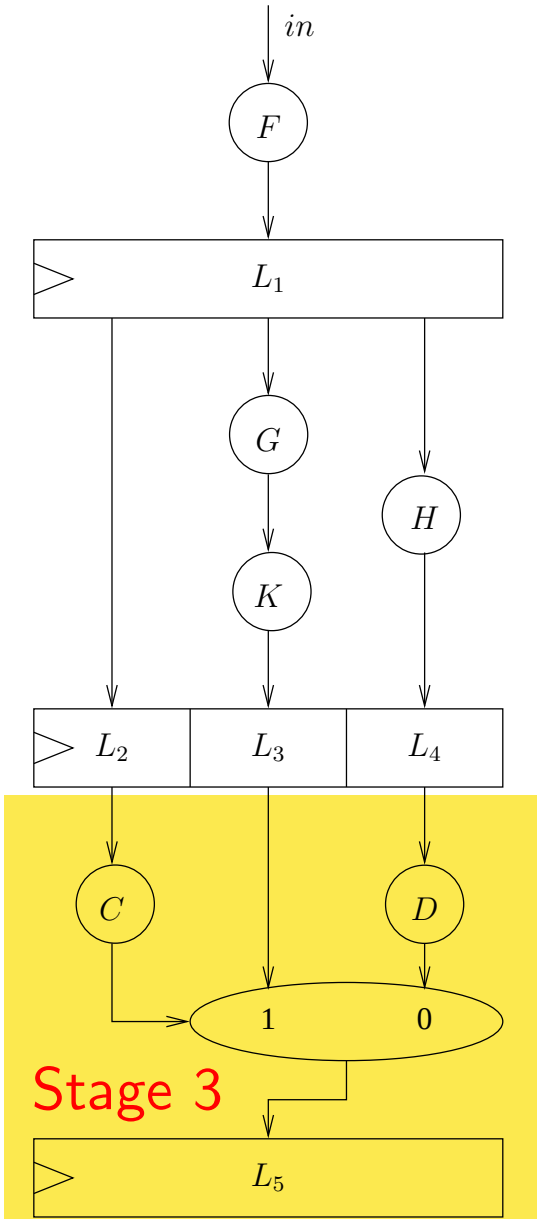
Example: Circuit Transformations



- A pipeline processes data in *stages*
- Data is processed in parallel – as in an assembly line
- Formal model:

$$
\begin{aligned}
L_1 &= f(I) \\
L_2 &= L_1 \\
L_3 &= k(g(L_1)) \\
L_4 &= h(L_1) \\
L_5 &= c(L_2)\,?\,L_3 \,:\, l(L_4)
\end{aligned}
$$

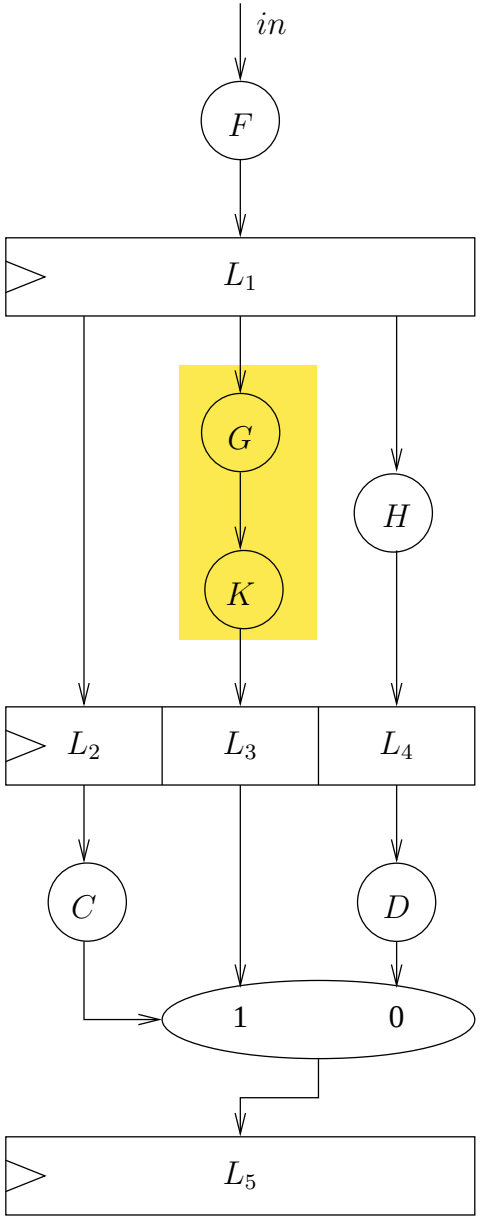
Example: Circuit Transformations



- A pipeline processes data in *stages*
- Data is processed in parallel – as in an assembly line
- Formal model:

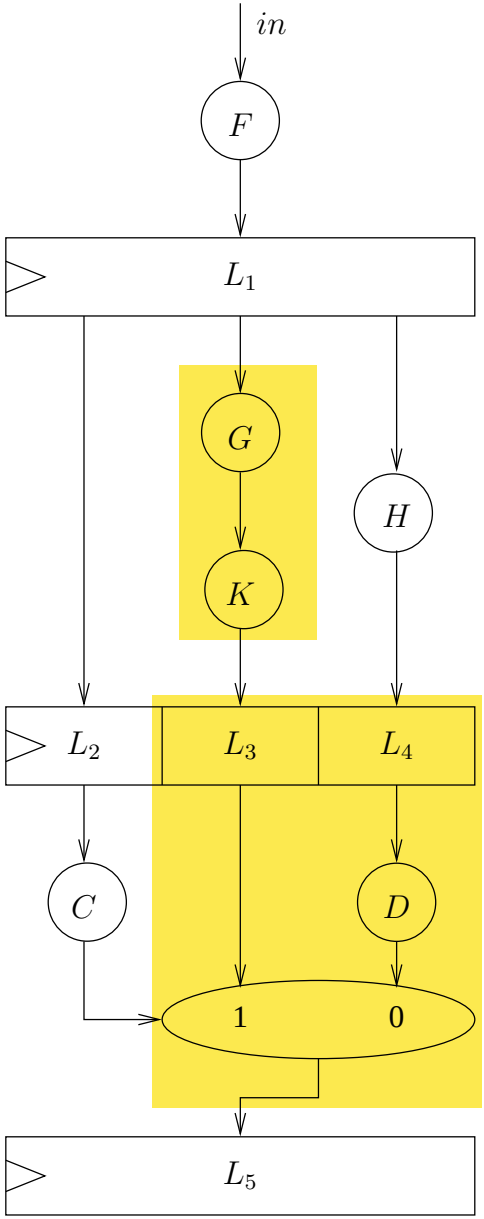
$$
\begin{aligned}
L_1 &= f(I) \\
L_2 &= L_1 \\
L_3 &= k(g(L_1)) \\
L_4 &= h(L_1) \\
L_5 &= c(L_2)\,?\,L_3\,:\,l(L_4)
\end{aligned}
$$

Example: Circuit Transformations



- The maximum clock frequency depends on the longest path between two latches
- Note that the output of g is used as input to k
- We want to speed up the design by postponing k to the third stage

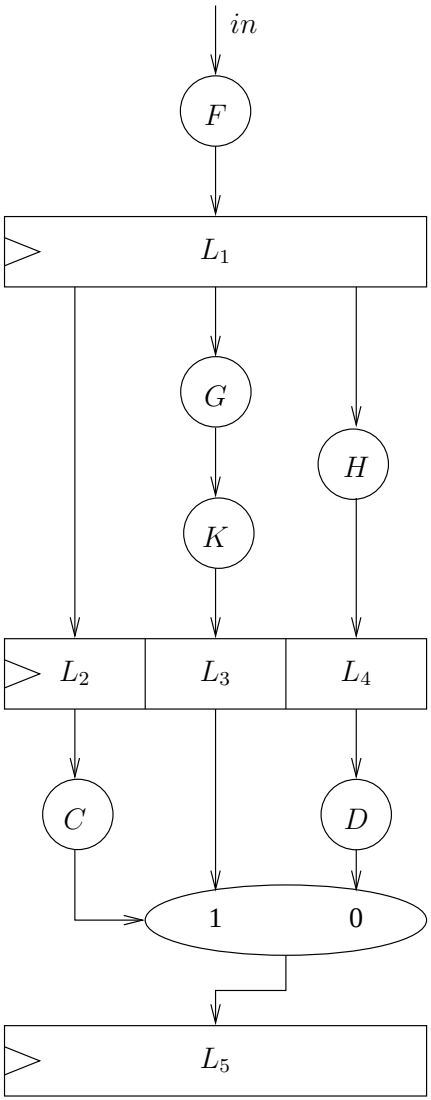
Example: Circuit Transformations

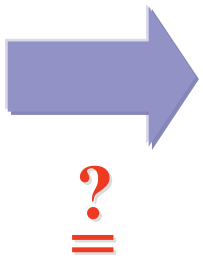


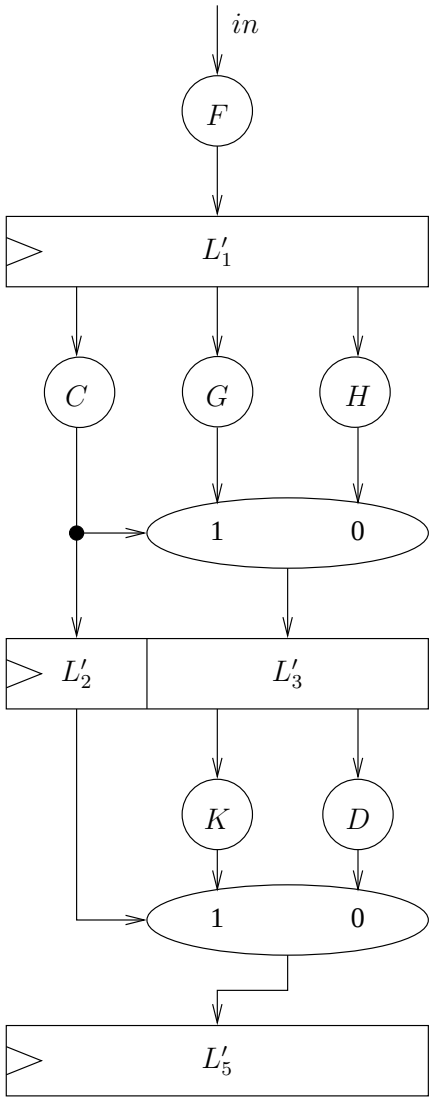
- The maximum clock frequency depends on the longest path between two latches
- Note that the output of g is used as input to k
- We want to speed up the design by postponing k to the third stage
- Also note that the circuit only uses one of L_3 or L_4, never both

$\Rightarrow$ We can remove one of the latches

Example: Circuit Transformations







Example: Circuit Transformations

$$
\begin{aligned}
L_1 &= f(I) \\
L_2 &= L_1 \\
L_3 &= k(g(L_1)) \\
L_4 &= h(L_1) \\
L_5 &= c(L_2)\,?\,L_3 \,:\, l(L_4)
\end{aligned}
\qquad
\begin{aligned}
L'_1 &= f(I) \\
L'_2 &= c(L'_1) \\
L'_3 &= c(L'_1)\,?\,g(L'_1) \,:\, h(L'_1) \\
L'_5 &= L'_2\,?\,k(L'_3) \,:\, l(L'_3)
\end{aligned}
$$

$$L_5 \stackrel{?}{=} L'_5$$

Example: Circuit Transformations

$$\begin{aligned} L_1 &= f(I) \\ L_2 &= L_1 \\ L_3 &= k(g(L_1)) \\ L_4 &= h(L_1) \\ L_5 &= c(L_2)\,?\,L_3 : l(L_4) \end{aligned}$$

$$\begin{aligned} L'_1 &= f(I) \\ L'_2 &= c(L'_1) \\ L'_3 &= c(L'_1)\,?\,g(L'_1) : h(L'_1) \\ L'_5 &= L'_2\,?\,k(L'_3) : l(L'_3) \end{aligned}$$

$$L_5 \stackrel{?}{=} L'_5$$

- Equivalence in this case holds regardless of the actual functions
- Conclusion: can be decided using *Equality Logic and Uninterpreted Functions*