Covalidation of Hardware-Software Systems

Ian G. Harris

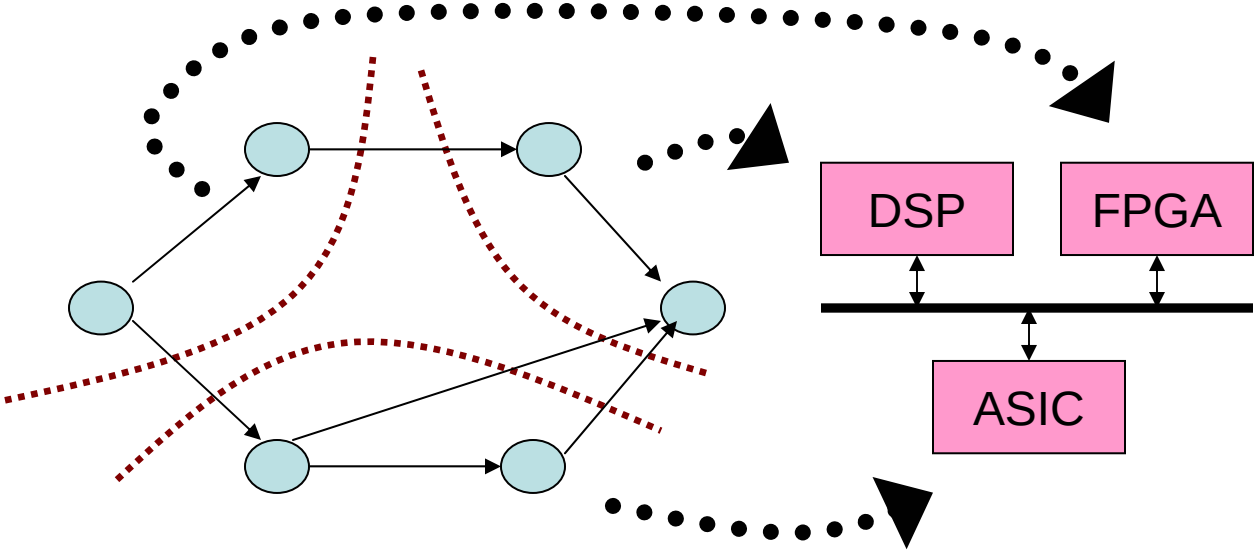
Department of Computer Science
University of California Irvine

Outline

- Hardware/Software Codesign and Covalidation
- Evaluation of Fault Models
- Timing Fault Model
- Automatic Test Generation
- Conclusions

Hardware/Software Codesign

```
int foo (int in1, int in2)
 int a, b, c;
 a = in1 + in2;
 b = 0; c = 0;
 while (c < a)
```



Behavioral Specification Task Graph Architecture

Specification: Describe the behavior of each task

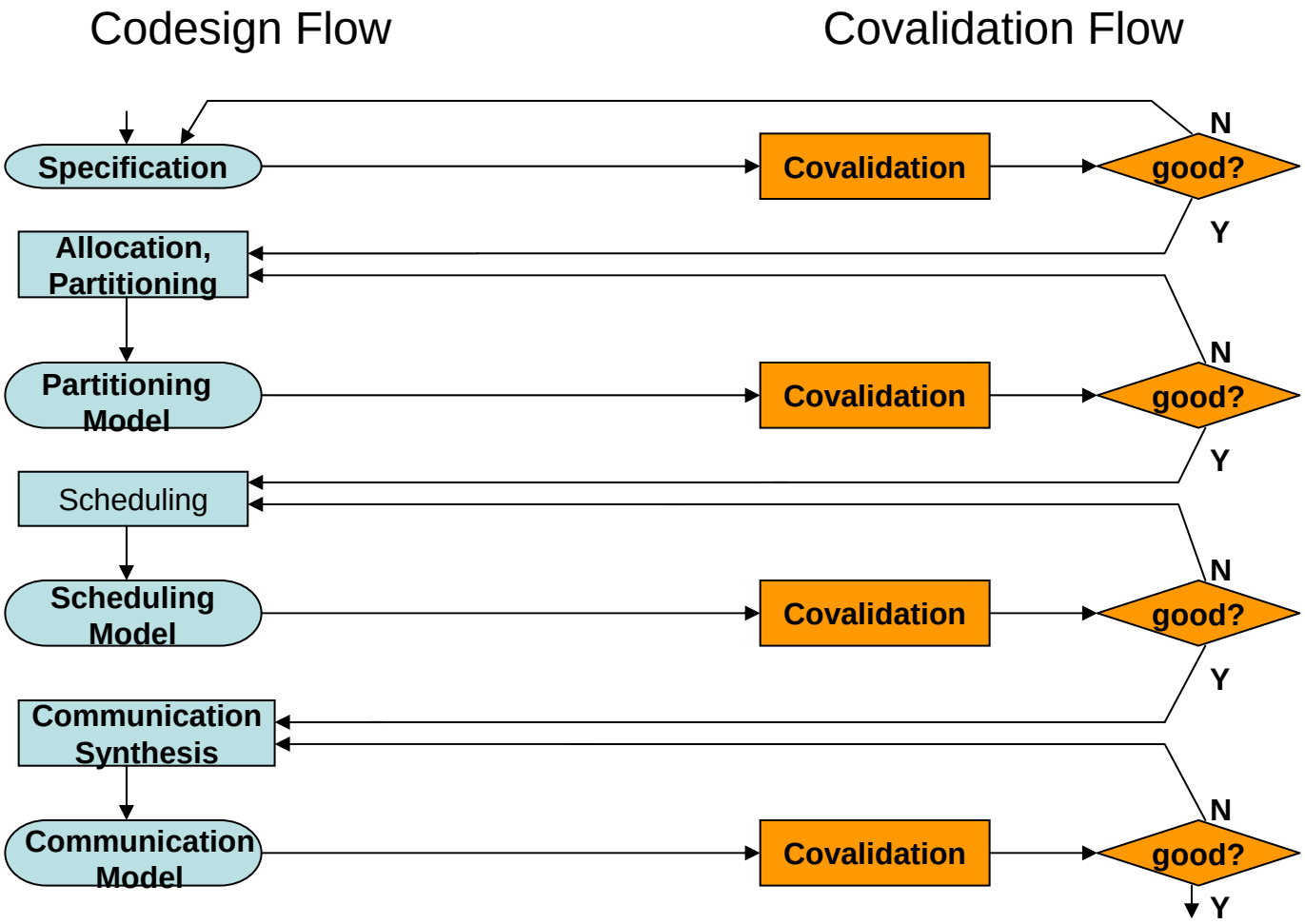
Partitioning: Group tasks to be implemented together

Allocation: Map partitions to hardware

Scheduling: Ordering the execution of tasks

Communication Synthesis: Map data transfers to comm. structures

Hardware-Software Codesign/Covalidation Flow



- **Design is refined at each step**
- **Each design step may introduce errors**

Importance of Covalidation

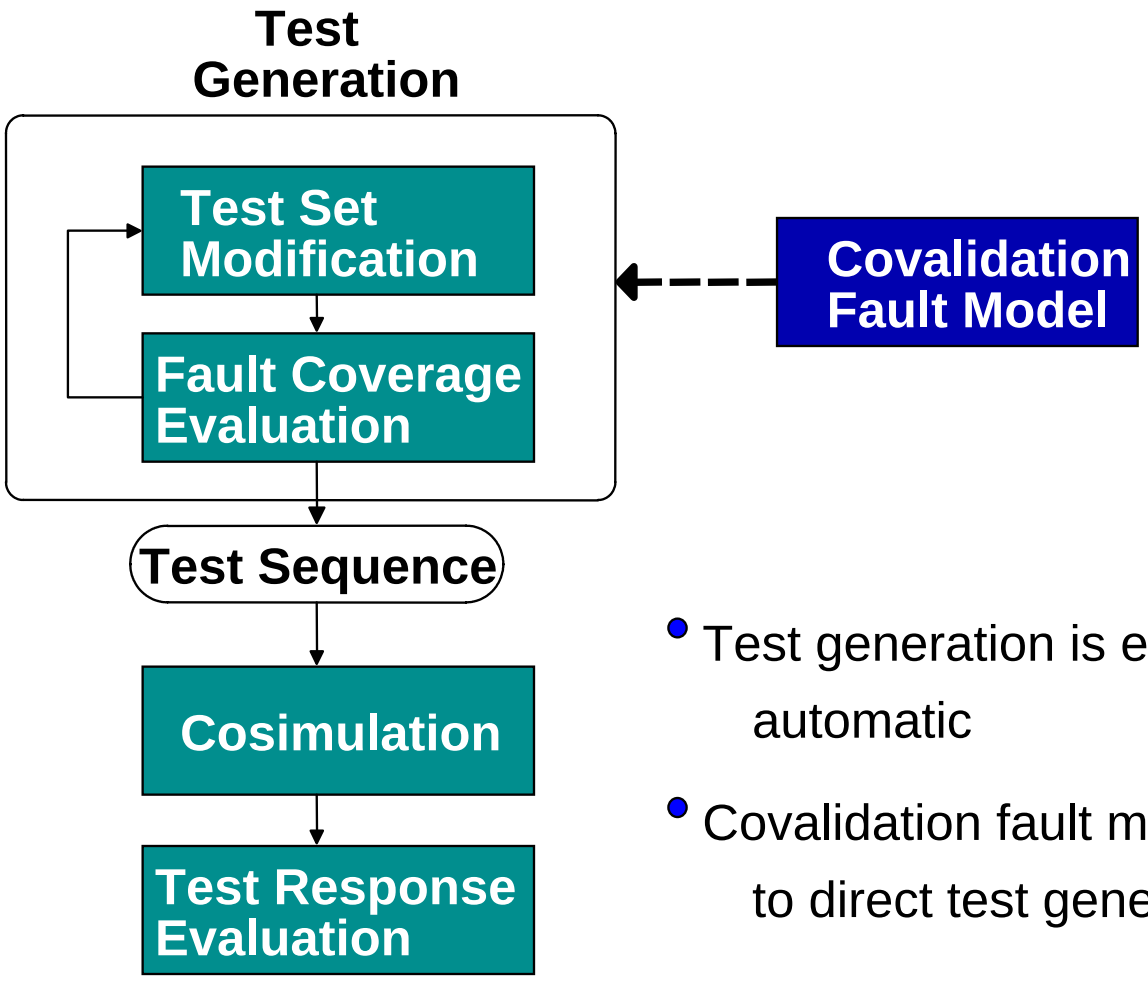
- **Validation/verification is a bottleneck in the design process**
  - High cost of design debug (designers, time-to-market)
  - High cost of faulty designs (loss of life, product recall)
- **Hardware/Software covalidation problem is more acute**
  - Hardware and software are often used together
  - Hardware and software are designed separately
  - Covalidation is performed late in the process, necessitating long redesign loops

General Verification Approaches

- **Formal verification**
  - Time complexity is high
  - Confidence is high for specified properties
- **Simulation based validation**
  - Full-chip validation
  - Confidence is lower
- Pentium 4 bugs found by FV (492) vs. Validation (5809) [1]

[1] B. Bentley, “Validating the Intel Pentium 4 Microprocessor,” DAC’01.

Stages of Covalidation



- Test generation is either manual or automatic
- Covalidation fault model is needed to direct test generation

Challenges in Hardware-Software Covalidation

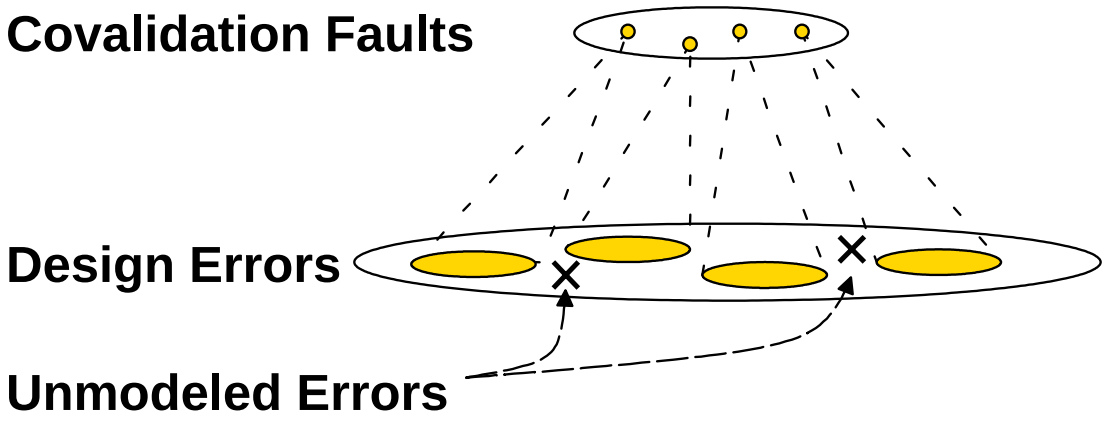
- **Covalidation Fault Model**
  - Describes a set of potential design defects
  - Provides goals for test generation
- **Automatic Test Generation**
  - Create a test sequence which guarantees detection of design defects
  - Allows the covalidation process to be more fully automated

Covalidation Fault Models

- A covalidation fault models the behavior of a set of design errors
- Properties of a covalidation fault model:

Modeling Accuracy

- Detection of all faults should imply the detection of a set of modeled faults



Model Size

- Few faults should model many design errors

Statement Coverage Example

- Assume that a defect is associated with a single statement in the behavioral description

int foo (int in1, int in2)

```
1. int a, b, c;
2. a = in1 + in2;
3. b = 0; c = 0;
4. while (c < a)
5.   c = c + in1;
6. if (c < in2)
7.   return (a + b);
8. else
9.   return (a + c);
```

Each line could have a defect.

- The defect associated with each line is non-specific
- Detection of the fault is assumed if the statement is executed

Fault Model Evaluation

Goal: To measure the accuracy of a fault model.

- Quantitative evaluation is essential for model development
  - Identify weaknesses in existing models
  - Build confidence in the use of a fault model
  - Avoid the fate of software test (use manufacturing test as a model)

Method for Fault Model Evaluation

- Compare *fault coverage* to *error coverage*
  - Fault coverage is a fast approximation of error coverage
  - Compute both fault and error coverage for many benchmarks and test sequences
  - Examine the *difference* between the fault and error coverages and the *standard deviation of the difference*

Error Coverage Computation

- A design error model is needed to compute error coverage
- Design Error Model Requirements
  - Must describe a well defined subset of real design errors
  - Must be small enough to be tractable
- Error coverage is computed by injecting design errors and performing simulation
  - Error is detected if the output of the correct behavior is different from the output of the erroneous behavior

Example: a = b * c (correct) a = **x** * c (incorrect)

- Error is not detected if c = 0 or if statement is not executed

Design Error Model

- "Goof" errors - simple typographical mistakes
  - Accounted for 12.7% of design errors found in the Pentium 4*
- Goof errors are described in research in *mutation analysis*

1. Arithmetic Operator Replacement - Each +, -, *, / is replaced by each other
2. Relational Operator Replacement - Each >, <, =, !=, … is replaced by each other
3. Variable Replacement - Each variable is replaced with each other variable of same type

*** B. Bentley, "Validating the Intel Pentium 4 Microprocessor", DAC 2001**

Evaluation Experiments

- We evaluate statement coverage and branch coverage
- We use three small examples written in Java

Benchmark	Statemts	Branches	Errors
GCD	19	1	162
Diffeq	21	7	502
TLC	65	14	214

- 20 random test sequences are used
- Each sequence achieves 80% - 100% fault coverage
  - Only high coverage values are interesting

Evaluation Result Summary

- Experiment demonstrates the type of information that can be gained from this evaluation technique
- Data is not sufficient to draw conclusions

	Statement Coverage		Branch Coverage	
Benchmarks	Average diff	St. Dev. diff	Average diff	St. Dev. diff
GCD	16.35	3.17	16.68	3.16
Diffeq	8.88	11.59	14.60	20.45
TLC	10.72	5.06	2.28	6.24

- St. dev. is more important than average
  - Low st. dev shows consistency/fidelity
- St. dev is roughly proportional to the number of errors

Development of Hardware/Software Fault Models

- **Must be formulated for a behavioral description**

  Manufacturing test has focused on logic level

- **Must consider both hardware and software features**

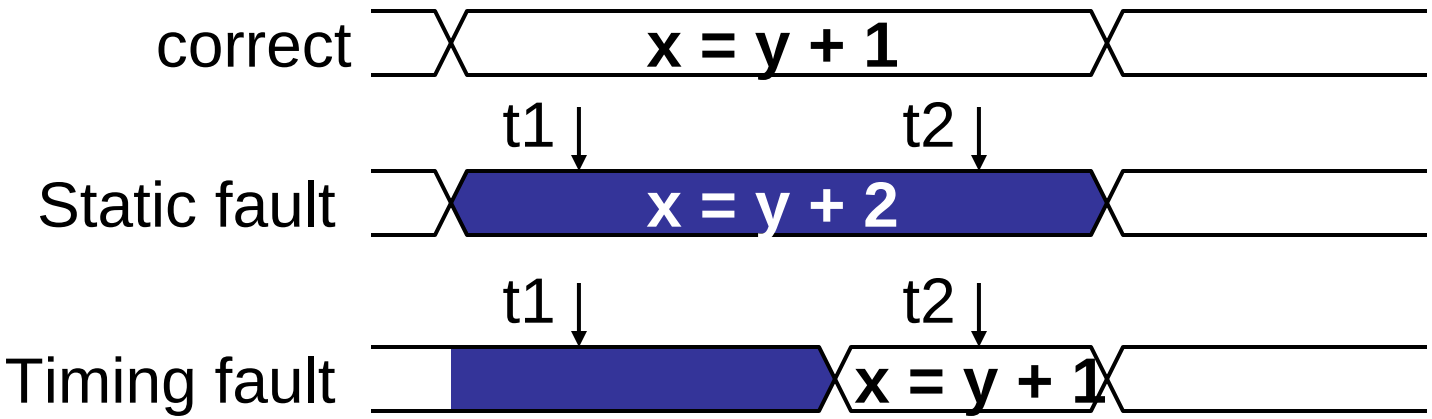
- **Hardware-Oriented Language Features:**

  Timing - signals vs. variables

  Concurrency - processes

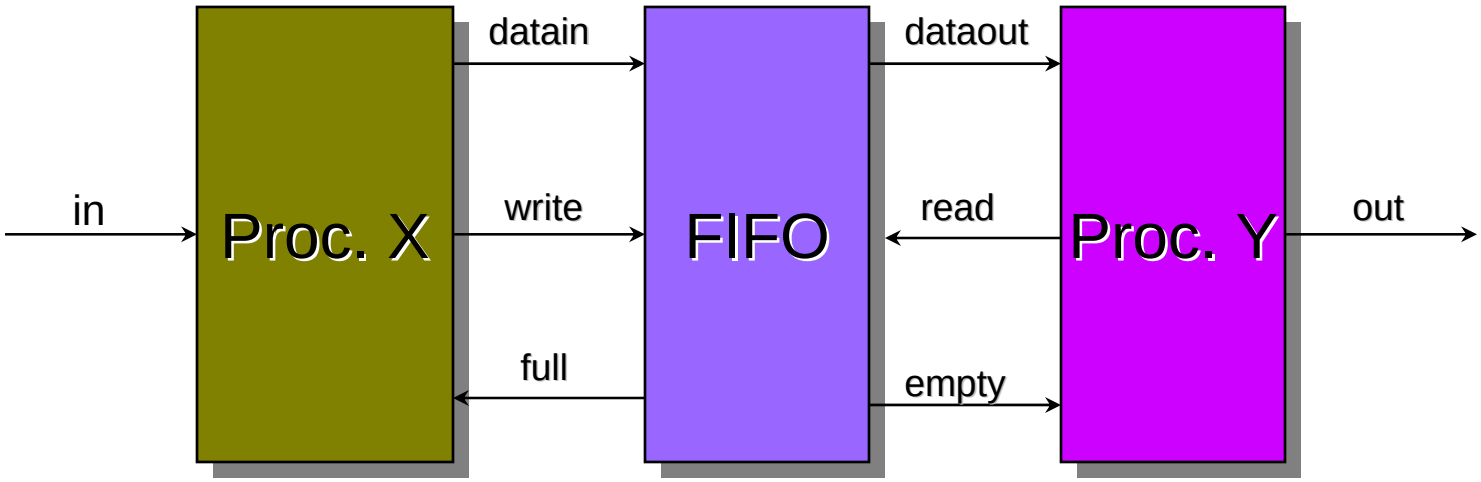
Static Faults vs. Timing Faults

- **Static faults**
  - Independent of absolute event timing
- **Timing faults**
  - Depends on a specific timing of events



Timing Fault Example

- Producer/Consumer with FIFO to allow rate mismatch
- Delay on “empty” signal impacts synchronization



Mis-Timed Event (MTE) Fault

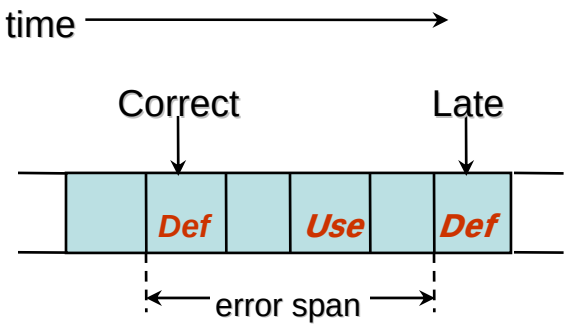
- Signal refs are classified as either **Definition** or **Use**
- Two types of MTE faults can occur
  - MTE_{early} – definition occurs earlier than the correct time
  - MTE_{late} – definition occurs later than the correct time

FIFO Description

Def -> empty <= 1

Proc. Y Description

Use -> if (empty = 0) then
p := ReadFromFIFO( ) ;

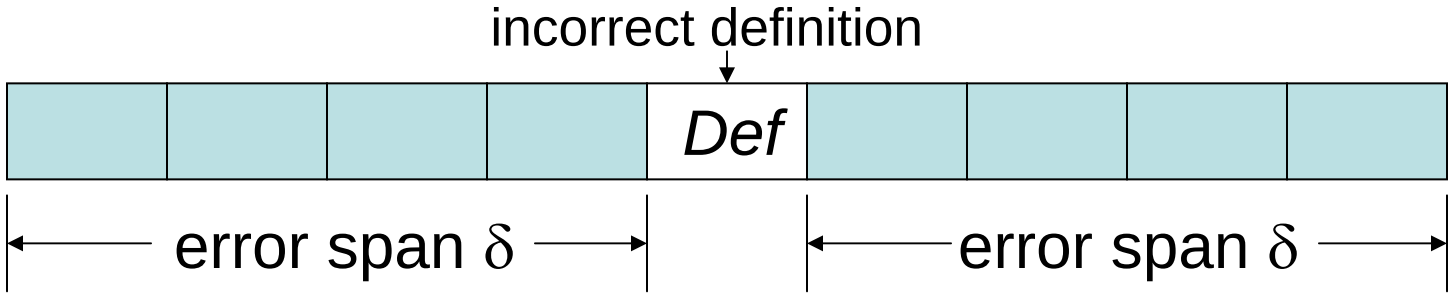


MTE Fault Detection

- All definition-use (*du*) pairs must be executed during testing
  - MTE early faults are detected by *du* pairs
  - MTE late faults are detected by *ud* pairs
- Time difference between the definition and use must not exceed the error span threshold δ
  - Def and Use must be close in time so that a small time variation will reorder the def and use.
  - Magnitude of δ determines sensitivity to timing variation

Error Span Threshold δ

- Determines the certainty that a def-use pair is reordered in the presence of a fault



- **Large δ: small change in def time may not reorder the def and use**
  - Testing requirements are less stringent, high coverage
- **Small δ: small change in def time is more likely to reorder def and use**
  - Testing requirements are more stringent, low coverage

MTE Coverage, Experimental Results

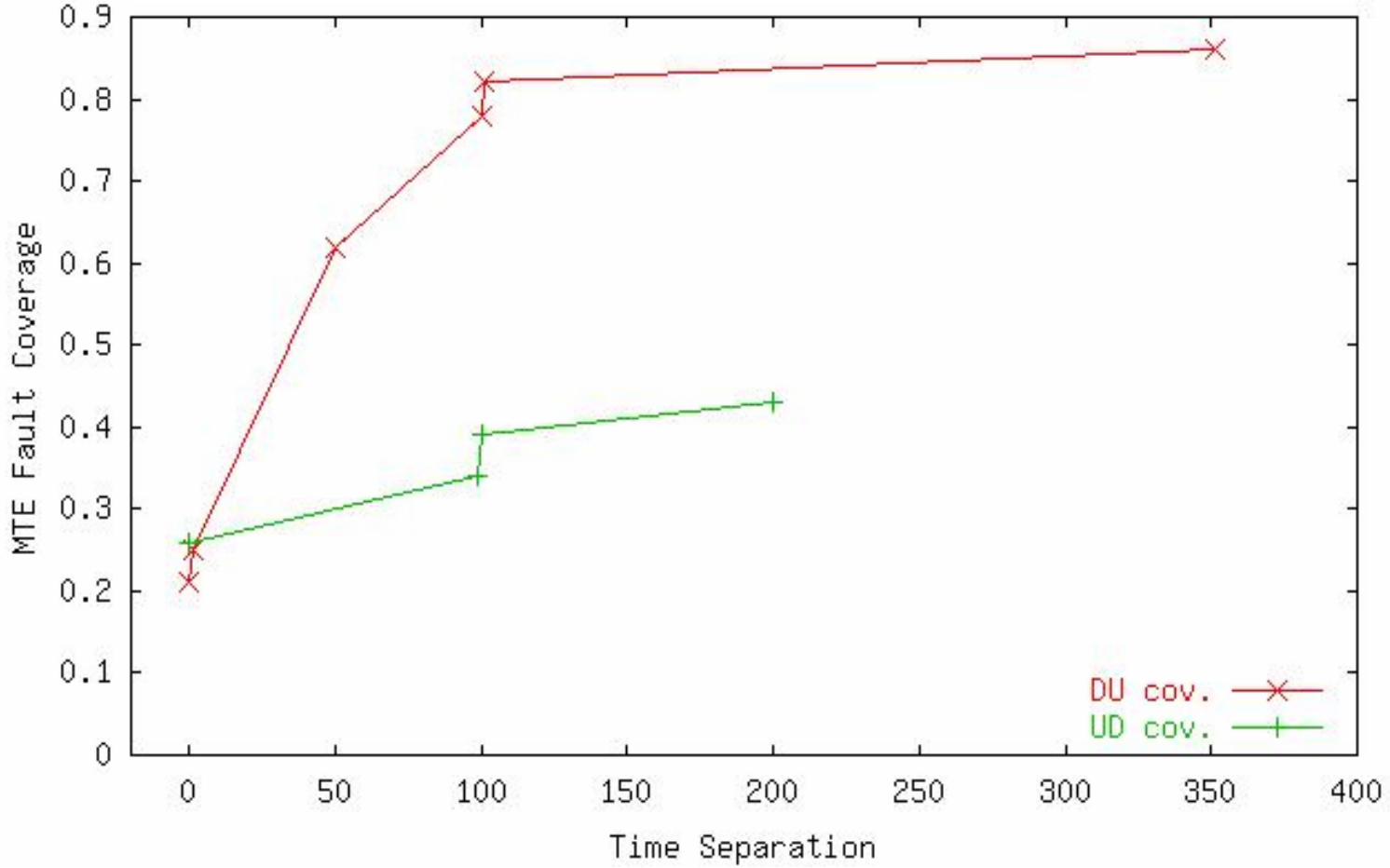
Bench-mark	# of stmt.	# of signals	# of pairs	**MTE cov. ***	Stmt. Cov.
AAL1	538	22	1732	**0.1**	0.70
switch	402	29	200	**0.65**	0.93
risc8	306	37	1032	**0.54**	0.60
DTMF	1257	17	262	**0.39**	0.54

- Industrial examples in Verilog
- Functional testbenches provided with each example

* Error span threshold is infinitely large. FC is maximized.

Impact of δ on MTE Coverage, Data Switch

MTE cov

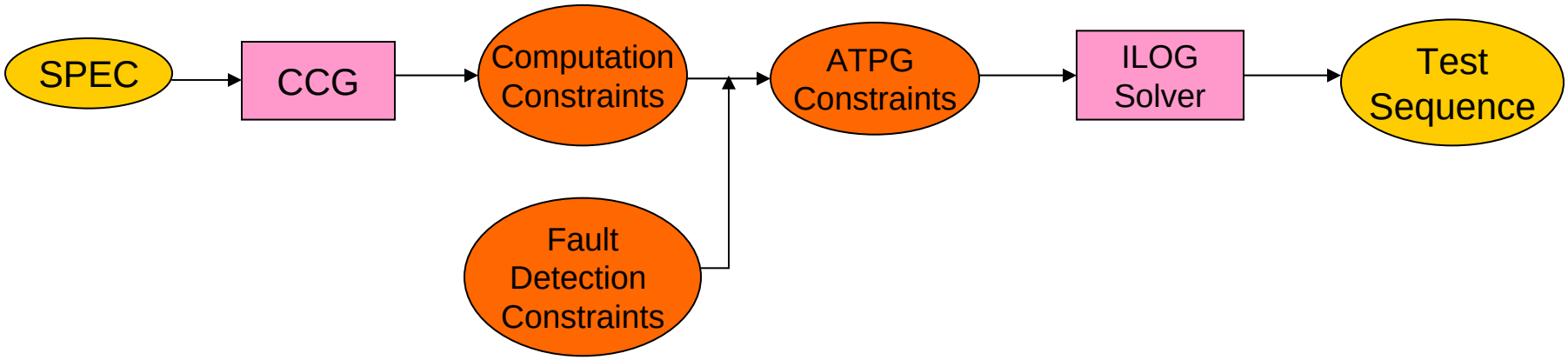


threshold δ

Automatic Test Generation

- Create test sequences to detect MTE faults
- Formulate the problem as a constraint logic programming (CLP) problem
- Use a generic CLP solver to perform test generation
- Solver searches the space of all computations to identify one which detects each fault

Test Generation Process

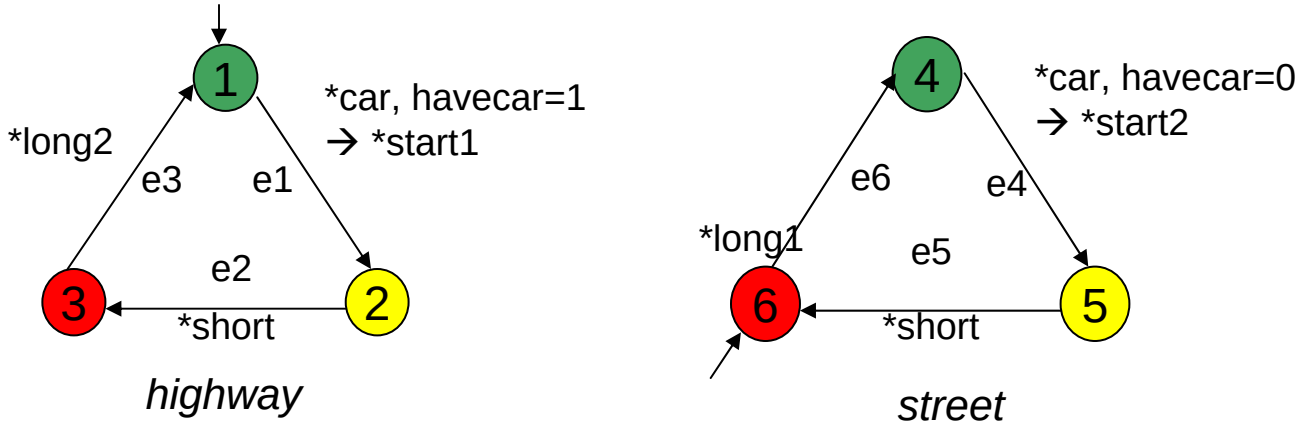


- Input: *a* behavioral system specification
- Output: test sequence to detect a timing fault
- *CCG*: Computation Constraints Generator
- *Computation Constraints* describe system behavior

Behavioral Representation: CFSM model

Co-design Finite State Machine: A system is defined as a network of CFSMs where each CFSM describes a concurrent process in the system. The CFSMs communicates via events on signals.

- An Example of CFSMs System: traffic light controller



- Each edge is a cause-reaction pair

- MTE_{early} fault on *short signal: asserted while the *highway* is in the *green* state

Computation Model

- Each computation of the system must be described by the values of a set of integer variables
- Computation Variables:

State Variable contains the value of state for each CFSM *c* at a given time step *t*. $SV_{highway,0}$ = “green”

Edge Variable the edge in each CFSM *c* which is traversed at a given time step *t*. $EV_{highway,1}$ = “e1”

Signal Variable the value of signal at a given time step.

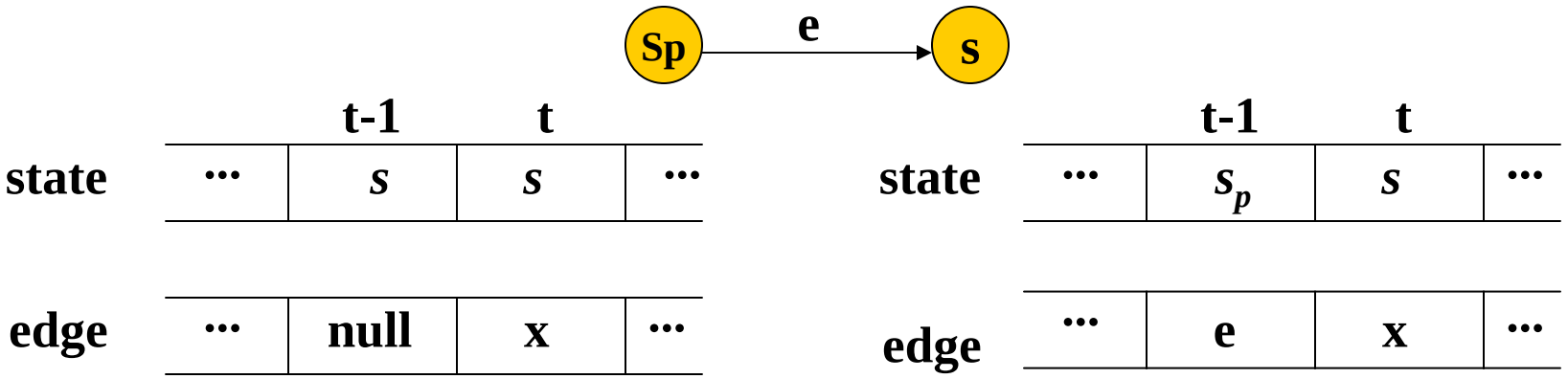
trigger signal *short_0 = 0

value signal havecar_2 = 1

State Constraints of CFSM Computation

a CFSM can be in state *s* at time *t* ($SV_{c,t} = s$) if one of the following statements is true:

- The CFSM is in state *s* at time *t-1* and the CFSM does not traverse an edge at time *t-1*;
- The CFSM is in a state s_p at time *t-1* and an edge from state s_p to s is traversed at time *t-1*.



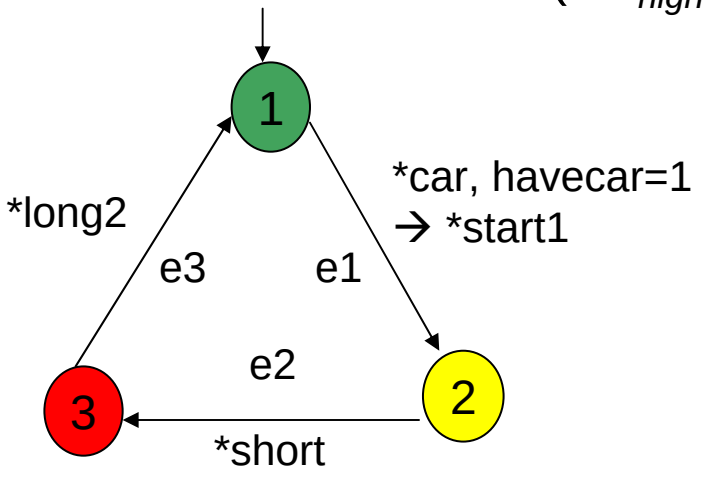
Example of State Constraints

Example: state constraints of CFSM *highway* at time step 2,

$(SV_{highway,2} = \text{“yellow”}) \rightarrow (SV_{highway,1} = \text{“yellow”}) \cap (EV_{highway,1} = \text{NULL}) \cup (SV_{highway,1} = \text{“green”}) \cap (EV_{highway,1} = e1)$

$(SV_{highway,2} = \text{“red”}) \rightarrow (SV_{highway,1} = \text{“red”}) \cap (EV_{highway,1} = \text{NULL}) \cup (SV_{highway,1} = \text{“yellow”}) \cap (EV_{highway,1} = e2)$

$(SV_{highway,2} = \text{“green”}) \rightarrow (SV_{highway,1} = \text{“green”}) \cap (EV_{highway,1} = \text{NULL}) \cup (SV_{highway,1} = \text{“red”}) \cap (EV_{highway,1} = e3)$

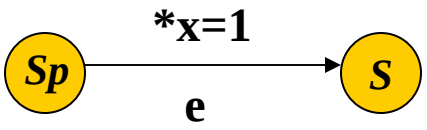


highway

Edge Constraints of CFSM Computation

a CFSM will traverse an edge e if all of the following statements are :

- The CFSM is in state s_p at time *t,* where s_p is the predecessor state of edge e;
- All of the trigger conditions of edge *e* are satisfied at time *t*.



	t
state	s_p
*x	1

	t
edge	e

Example of Edge Constraints

Example: edge constraints of CFSM *highway* at time 1,

$(EV_{highway,1} = \text{“e1”}) \rightarrow$

$(SV_{highway,1} = \text{“green”}) \cap (car_1 = 1) \cap (havecar_1 = 1)$ (1)

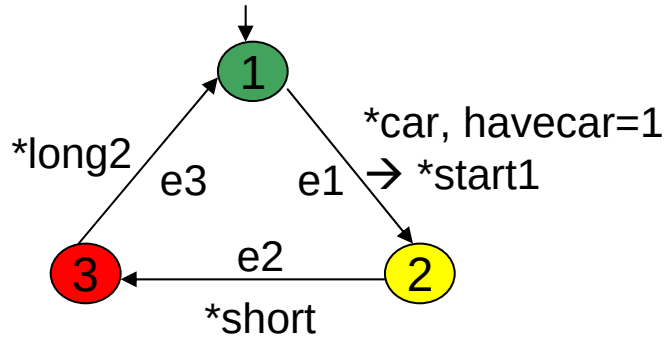
$(EV_{highway,1} = \text{“e2”}) \rightarrow$

$(SV_{highway,1} = \text{“yellow”}) \cap (short_1 = 1)$ (2)

$(EV_{highway,1} = \text{“e3”}) \rightarrow$

$(SV_{highway,1} = \text{“red”}) \cap (long2_1 = 1)$ (3)

$(EV_{highway,1} = \text{“null”}) \rightarrow NOT1 \cap NOT2 \cap NOT3$



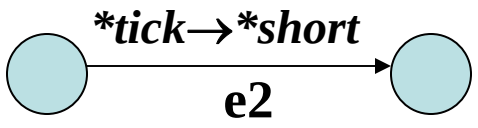
Signal Constraints of CFSM Computation

two types of signals exist in CFSM system

- *Trigger Signal*: such as **short*

  $*tsig_t = 1$ if at least one edge emitting this signal is traversed at time (t-1);

  $*tsig_t = 0$ otherwise.



	t-1		
edge	•••	**e2**	•••

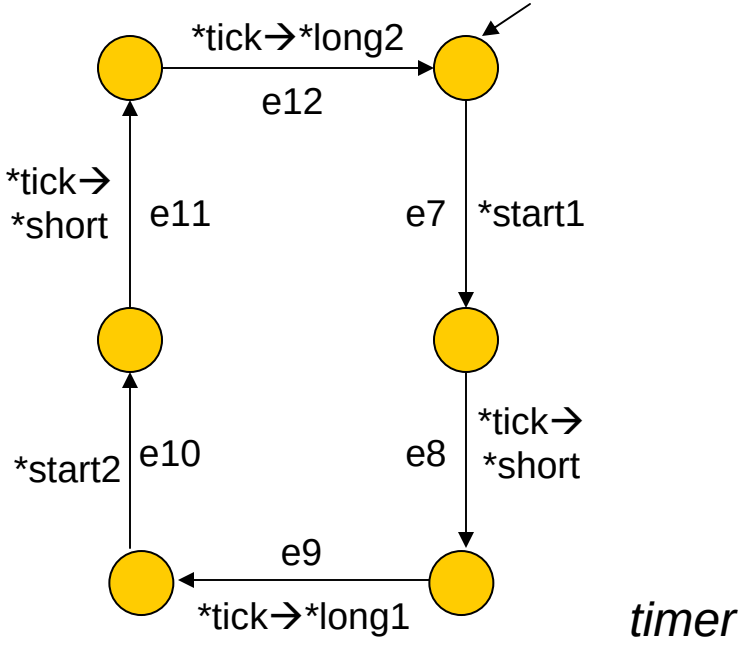
	t		
***short**	•••	**1**	•••

Example of Trigger Signal Constraints

Example: trigger signal **short* constraints of CFSM *highway* at time 3,

$(short_3 = 1) \rightarrow (EV_{timer,2} = \text{“e8”}) \cup (EV_{timer,2} = \text{“e11”})$

$(short_3 = 0) \rightarrow (EV_{timer,2} \neq \text{“e8”}) \cap (EV_{timer,2} \neq \text{“e11”})$

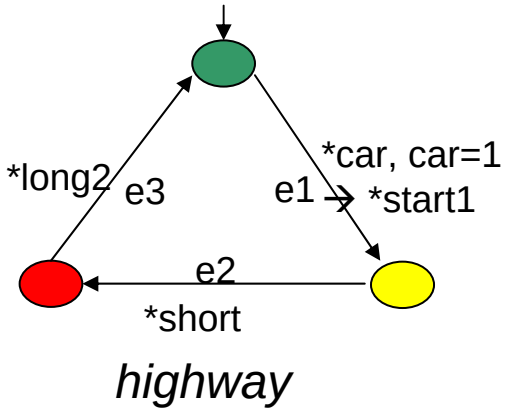


Formulation of Fault Detection Constraints

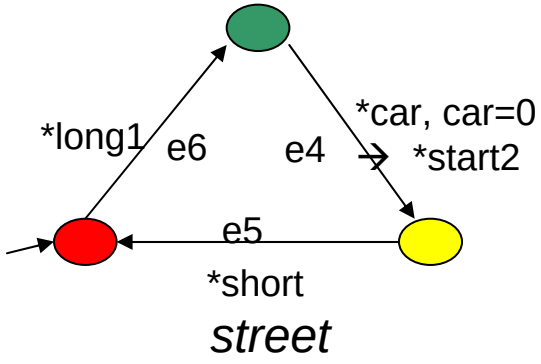
- *Fault Detection Constraints*: i.e.*short occurs early

time step	0	1	2
highway state	-	green	… …
**short*	-	1	… …

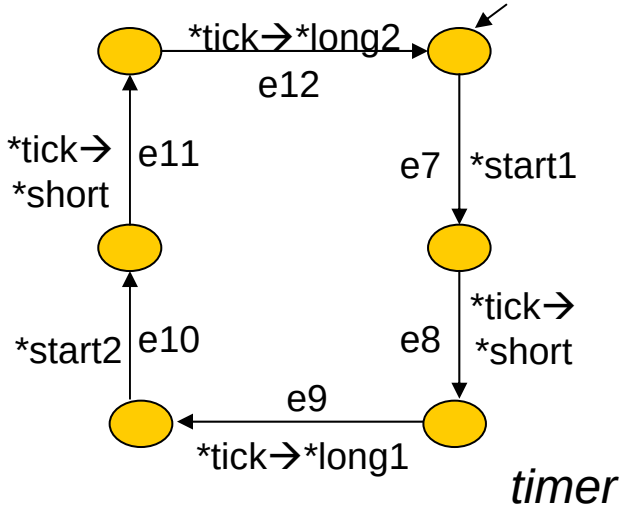
- *Equations:* $SV_{highway,1}$ = green, $short_1$ = 1.



highway



street



timer

Test Generation Result, Traffic Light Controller

- *Fault*: *short occurs early when highway is green
- *Fault Detection Constraints*: equations given earlier
- *Environment:*

  Machine P4, 2GHz CPU, 256MB MEM

  GNU-Prolog version1.2.1
- *Performance*: 45 ms

CFSM Computation

time step	0	1	2
highway state	-	green	-
**short*	-	1	
street state	red	red	red
highway state	green	green	yellow
*signal *short*	0	1	0

*tick	1	1	1
*car	0	1	0
car	0	1	1

Summary

- A method to evaluate the accuracy of a fault model
- A fault model for timing and synchronization errors
- A CLP-based test generation tool