Integration Testing through Reusing Unit Test Cases for Medical Software

Youngsul Shin

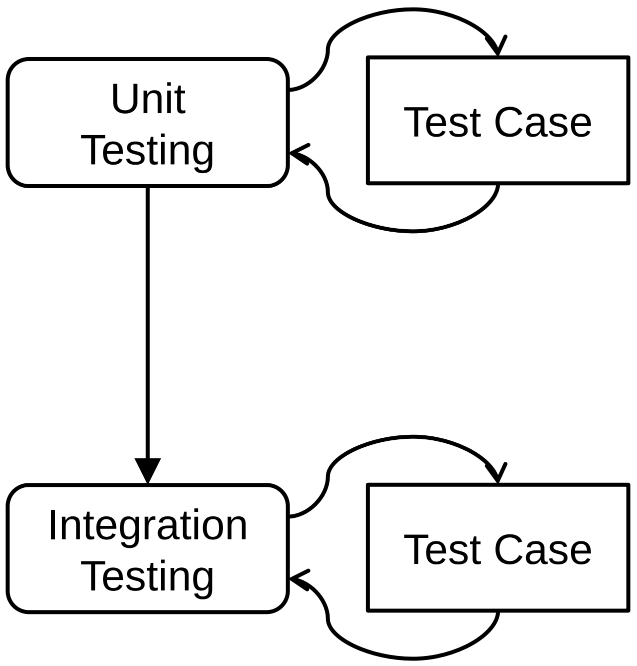
November 23, 2017

GYWELL Inc.

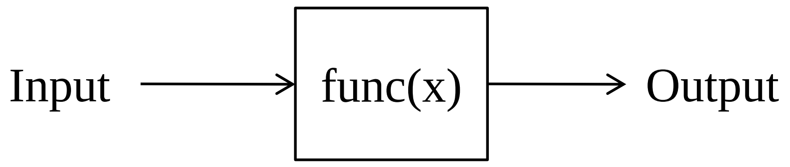
Contents

Introduction ▶ Motivation

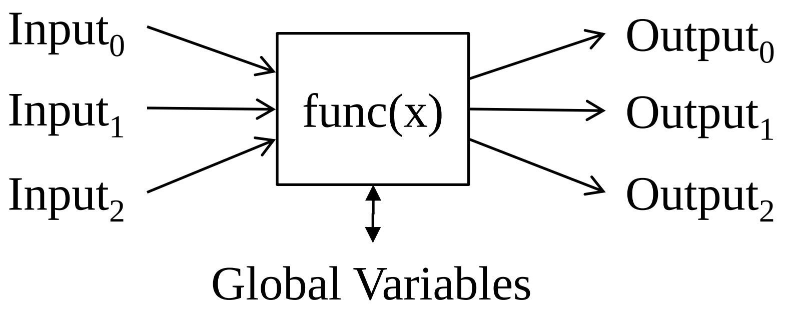
- Test cases defined at the unit testing are thrown out
- Difficult to define significant test cases at the integration testing
- Testing diverse aspects of complicated software



Newly Written Test Cases



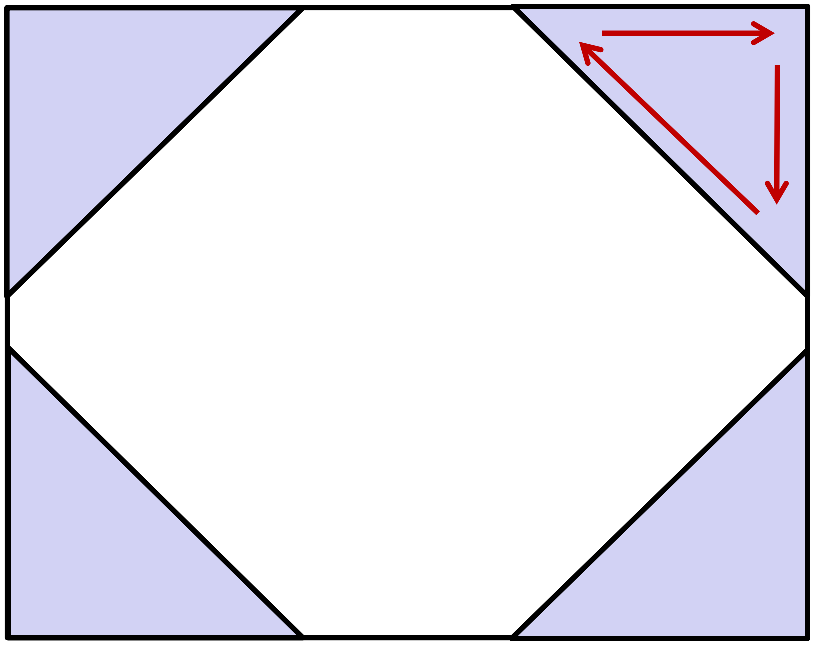
Approach with Low Coverage



Approach to Consider Function Characteristics

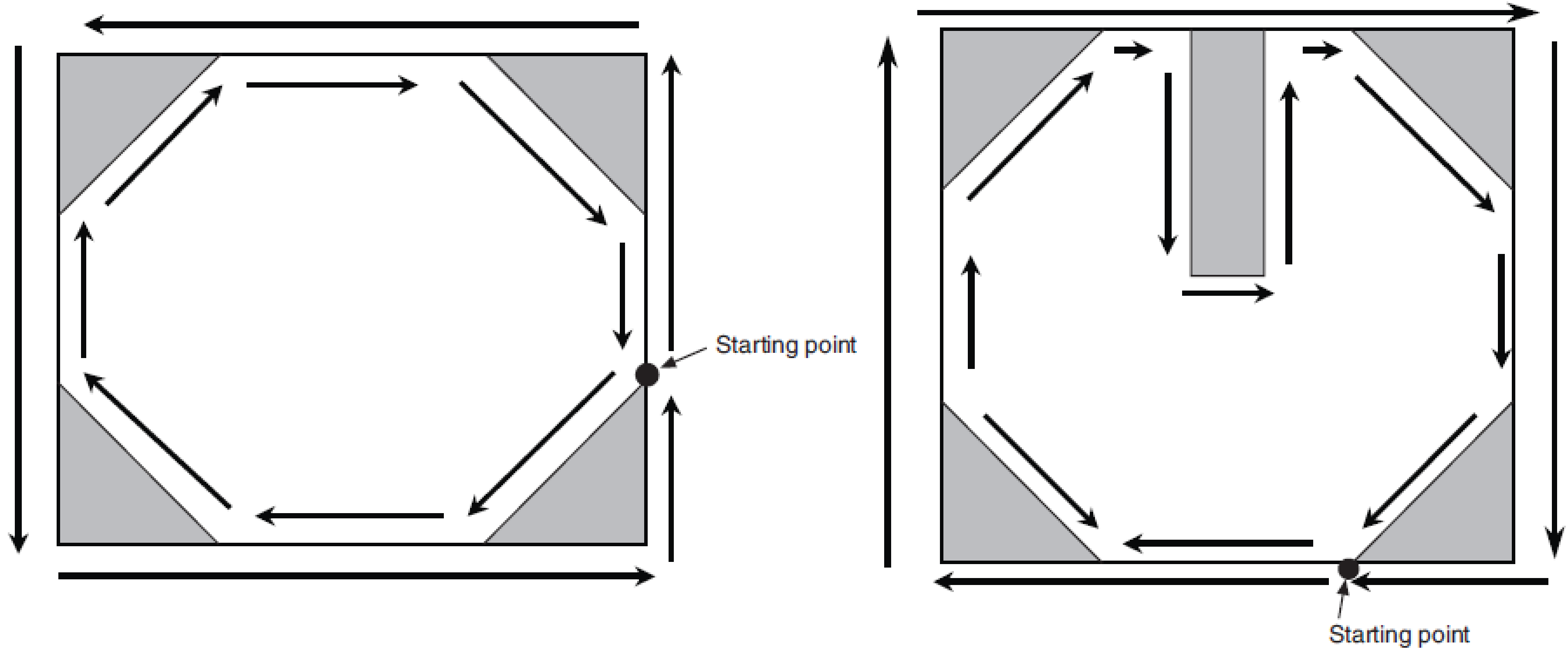
National Cancer Institute, Panama City

- Shielding blocks to protect healthy issue from the radiation
  - Four shielding blocks allowed
  - Input by drawing blocks



National Cancer Institute, Panama City (cont.)

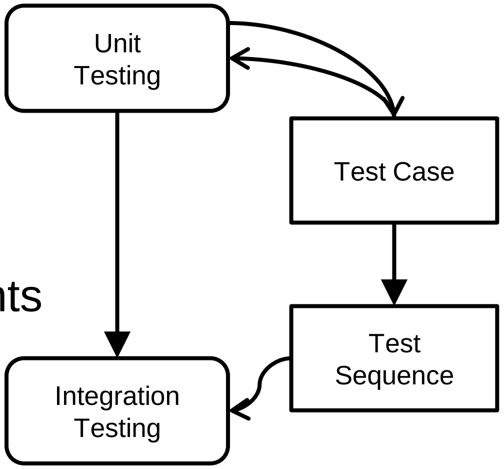
- Doctors wish to use five blocks
  - Drawing all five blocks as a single large loop



IAEA, “Investigation of an Accidental Exposure of Radiotherapy Patients in Panama,” Report of a Team of Experts, June 2001.

Introduction ▶ Contribution

- Reusing test cases that are defined at the unit testing
  - Writing a test case in JUnit
  - Mapping the test case onto an interface model
  - Gives significant test cases with high coverage to a tester
- Automatic generation of a test sequence
  - A function can be executed as many times as a tester wants
  - Time-efficient test sequence
  - It can apply to all kinds of interface models

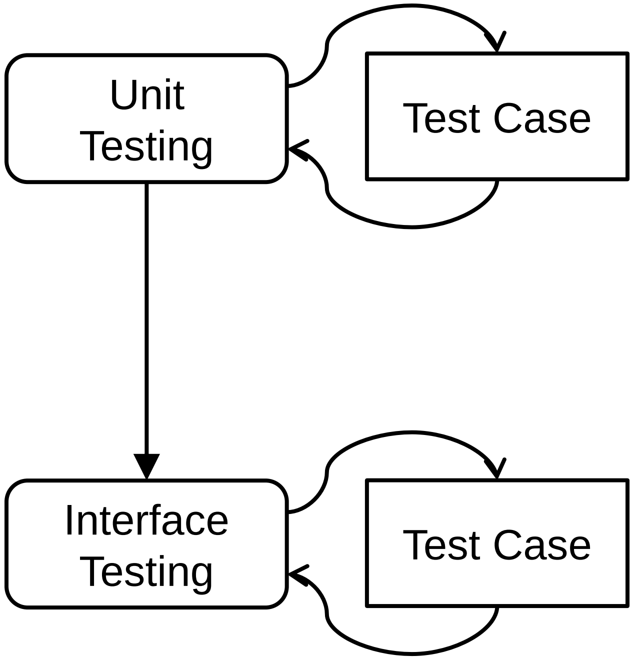


Reusing Test Cases

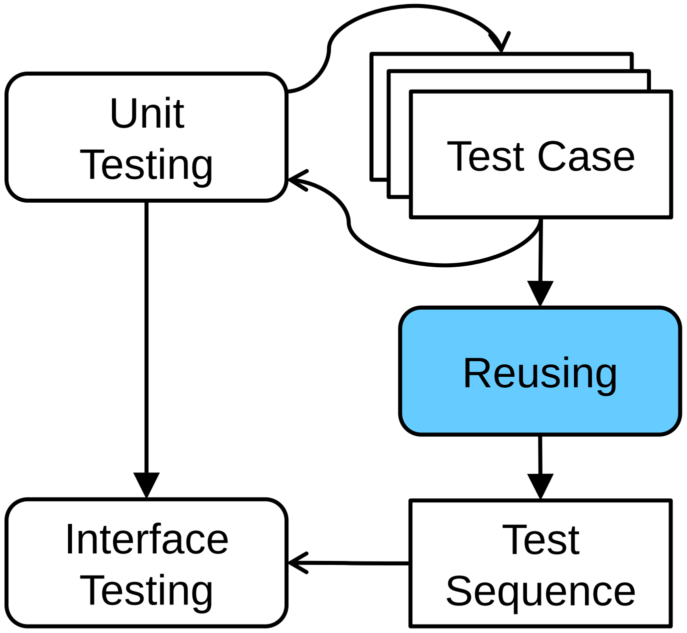
- A tester is given a test sequence to inspect diverse execution paths

Reusing Test Case

- Test cases for the unit testing are forgotten
- A tester writes new test cases

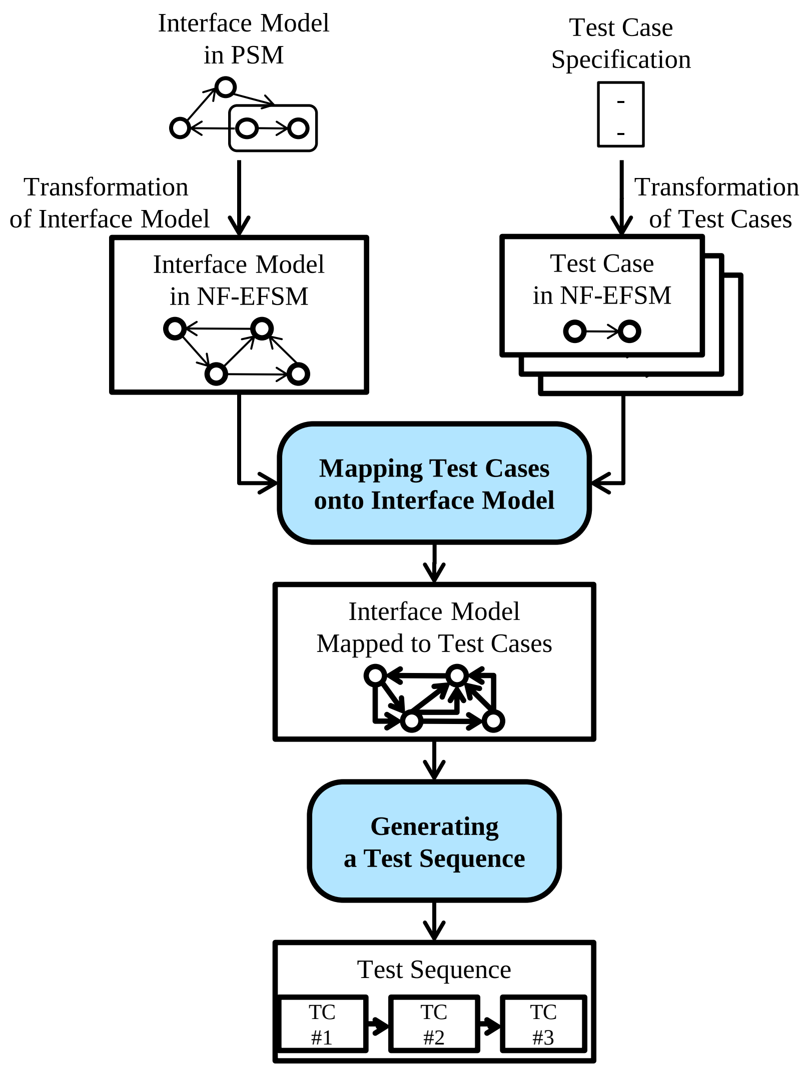


Newly Written Test Cases



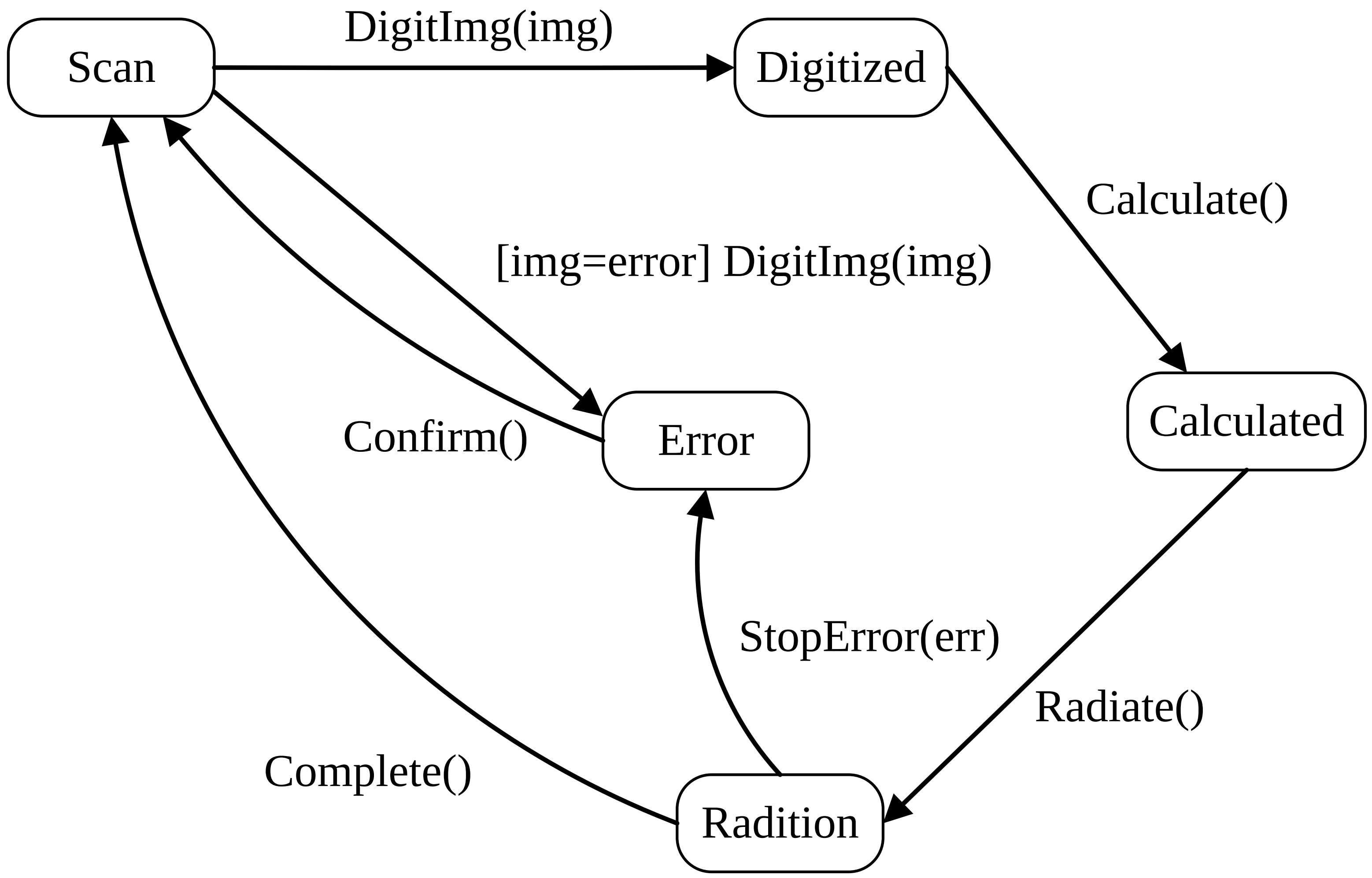
Reusing Test Cases

Approach for Test Sequence Generation


Interface Model in PSM
Test Case Specification
Transformation of Interface Model
Transformation of Test Cases
Interface Model in NF-EFSM
Test Case in NF-EFSM
Mapping Test Cases onto Interface Model
Interface Model Mapped to Test Cases
Generating a Test Sequence
Test Sequence
TC #1
TC #2
TC #3

- Transforming the model
  - Flattening the state machine
- Mapping test cases
  - State recognition
- Generating a test sequence
  - Greedy algorithm

Radiation Therapy Software



Mapping onto the Interface Model (cont.)

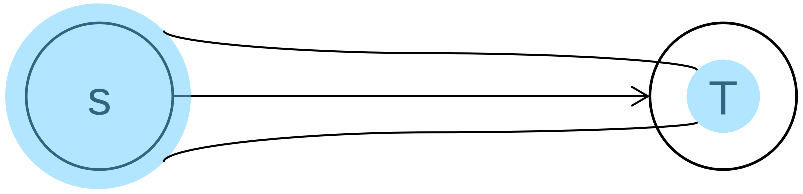
- Test case mapping
  - State recognition
- Model $M = (S, s_0, C, \sigma_0, P, I, O, T)$
- Test case $M' = (S', s_0', C', \sigma_0', P', I', O', T')$
- Mapping Rule

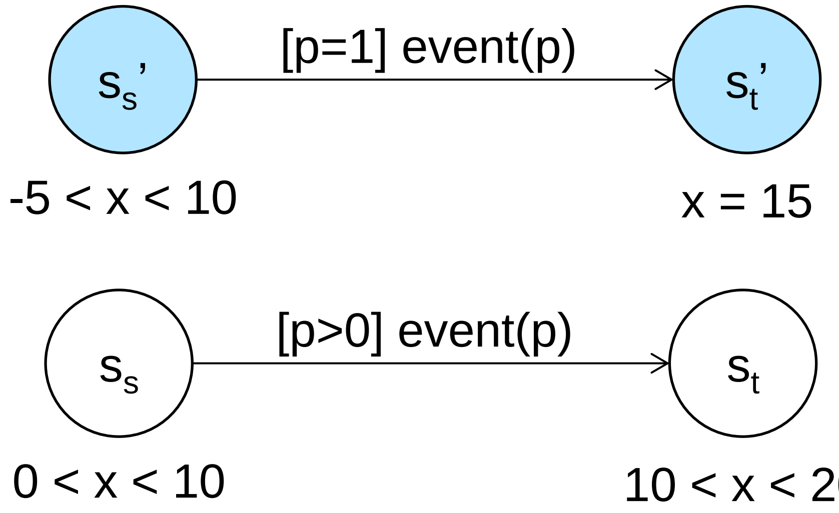
$A\,Trigger\ i\ of\ G_I\ is\ identical\ with\ a\ trigger\ i'\ of\ G'_{I'}$

$dom(s_s) \subseteq dom(s_s')$

$dom(g_{P^i}) \supseteq dom(g'_{P'^{i'}})$

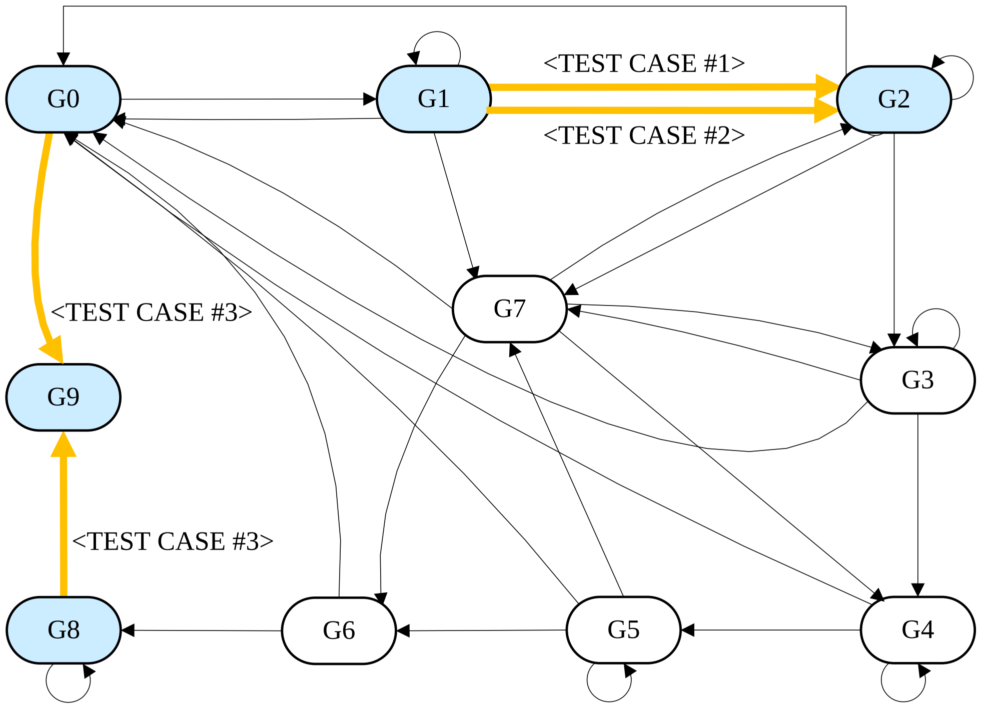
$dom(s_t) \supseteq dom(s_t')$





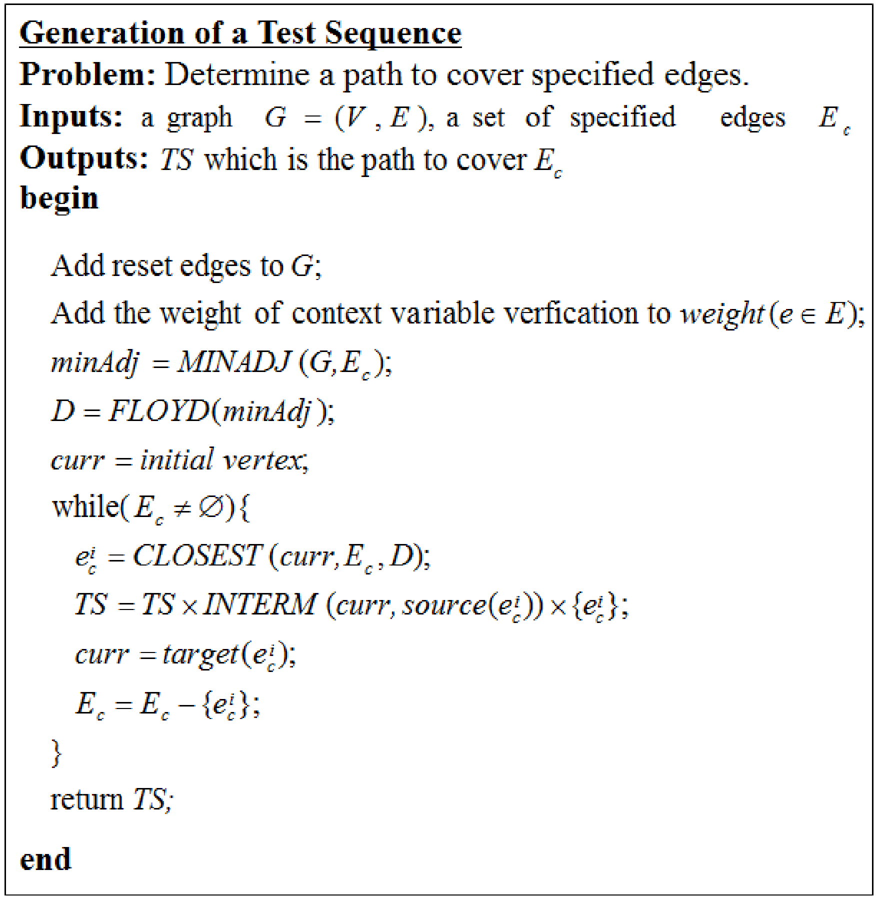
Mapping onto the Interface Model (cont.)

Test Case Number	Description	
1	Function	ld(pld)
	Precondition	vld=vpd=vbr=vdl=vli=warn=in=po=0
	Parameter Value	pld=0.6
	Return Value	n/a
	Postcondition	0.5<vld<50, vpd=vbr=vdl=vli=warn=in=po=0
2	Function	ld(pld)
	Precondition	vld= vpd=vbr=vdl=vli=warn=in=po=0
	Parameter Value	pld=49.9
	Return Value	n/a
	Postcondition	0.5<vld<50, vpd=vbr=vdl=vli=warn=in=po=0
3	Function	poweroff
	Precondition	po=null | po=0
	Parameter Value	n/a
	Return Value	n/a
	Postcondition	po=1



Time-Efficient Test Sequence Generation

- Greedy Test Sequence Generation Algorithm (GTS Algorithm)



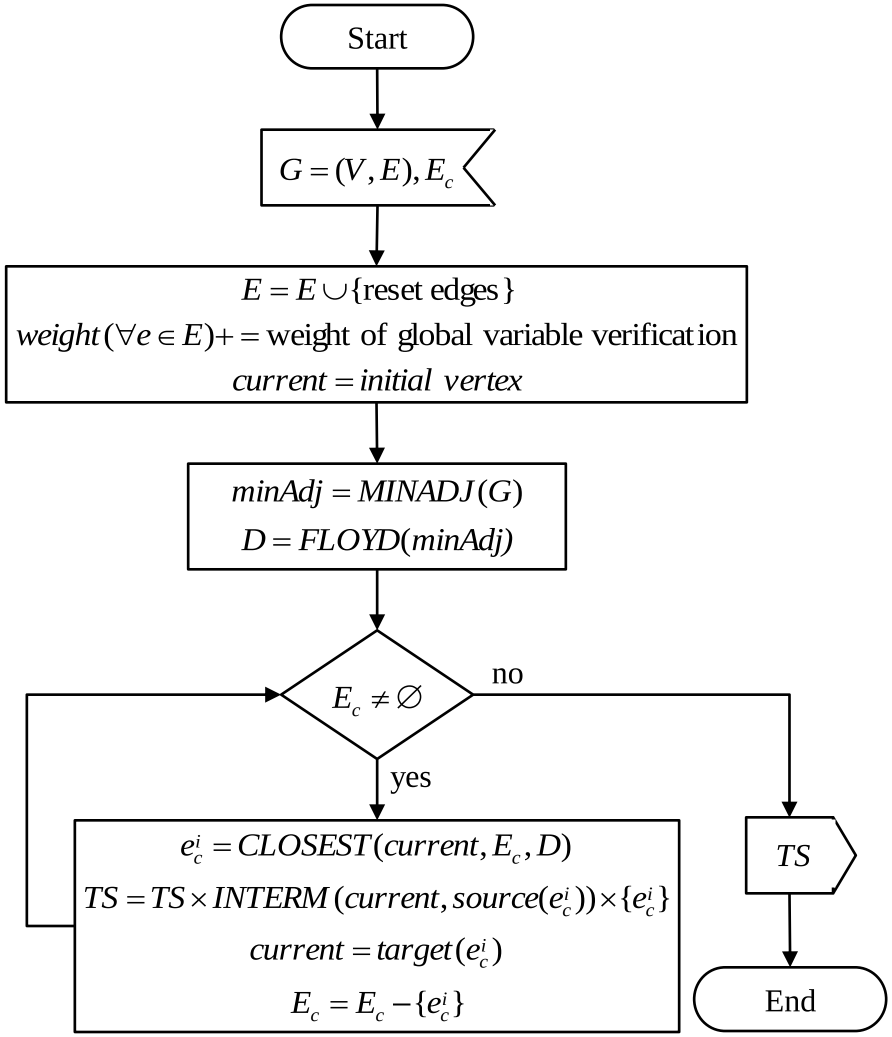
Generation of a Test Sequence

Problem: Determine a path to cover specified edges.

Inputs: a graph $G = (V, E)$, a set of specified edges E_c

Outputs: TS which is the path to cover E_c

```
begin
    Add reset edges to $G$;
    Add the weight of context variable verfication to $weight(e \in E)$;
    $minAdj = MINADJ(G, E_c)$;
    $D = FLOYD(minAdj)$;
    $curr = initial\ vertex$;
    while($E_c \neq \varnothing$){
        $e_c^i = CLOSEST(curr, E_c, D)$;
        $TS = TS \times INTERM(curr, source(e_c^i)) \times \{e_c^i\}$;
        $curr = target(e_c^i)$;
        $E_c = E_c - \{e_c^i\}$;
    }
    return $TS$;
end
```


Start
$G = (V, E), E_c$
$E = E \cup \{\text{reset edges}\}$
$weight(\forall e \in E) += \text{weight of global variable verification}$
$current = initial\ vertex$
$minAdj = MINADJ(G)$
$D = FLOYD(minAdj)$
$E_c \neq \varnothing$
no
yes
$e_c^i = CLOSEST(current, E_c, D)$
$TS = TS \times INTERM(current, source(e_c^i)) \times \{e_c^i\}$
$current = target(e_c^i)$
$E_c = E_c - \{e_c^i\}$
TS
End

Time-Efficient Test Sequence Generation

- Advantage of the GTS Algorithm

- generates a sequence to execute one edge several times

- quickly finds a solution close to optimal
  - Greedy approach
  - applies to general kinds of behavioral models
    - not try to find optimal paths
    - RCP problem is NP-complete, for the most general case

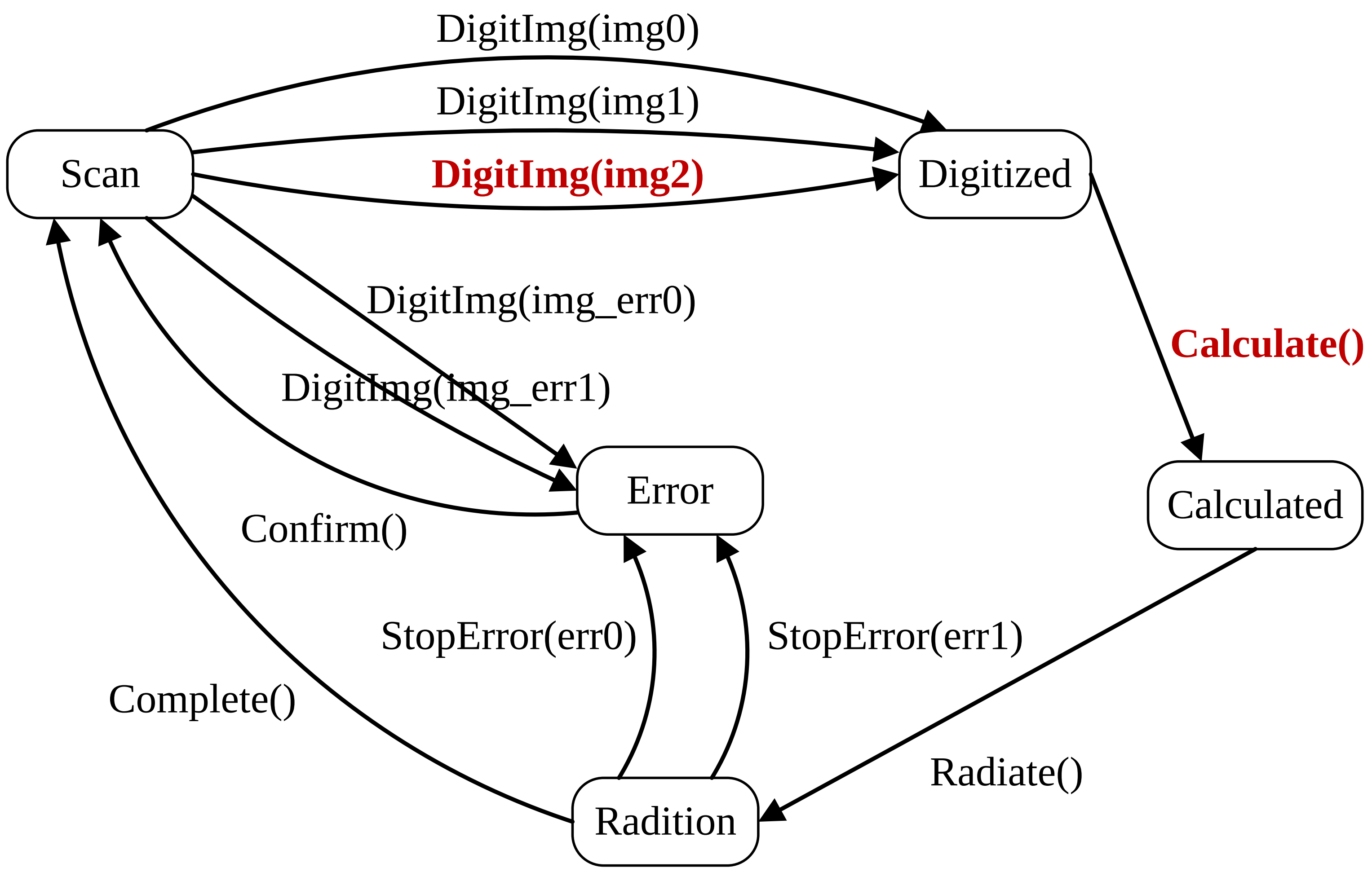
Experiments and Evaluation

- **Fault Detection Capability**
  - Faults occurred by interaction of functions
  - Reusing test cases defined at the unit test phase
  - Multiple test cases mapped on one transition
  - Various values of global variables & function parameters considered

- Test Coverage
  - As high as the unit test coverage

Experiments and Evaluation (cont.)

- Radiation Therapy Software

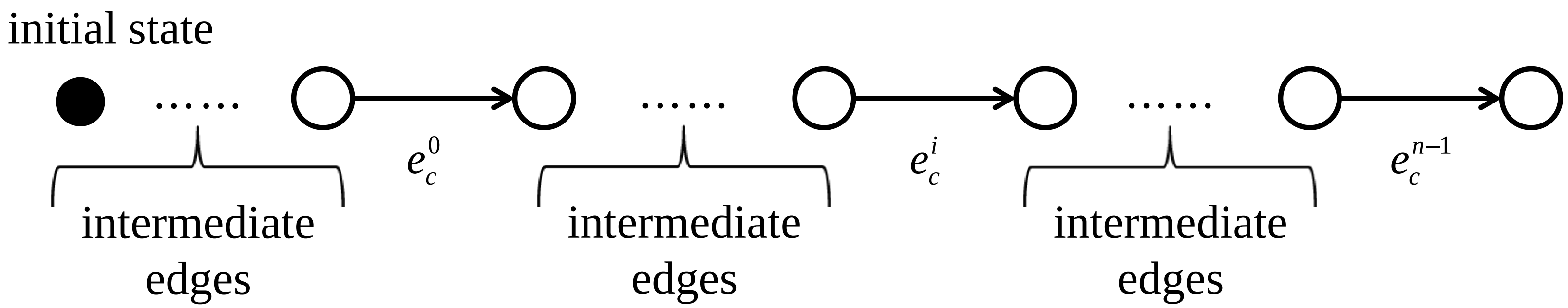


Experiments and Evaluation (cont.)

- Performance of the GTS algorithm
  - quickly finds a solution close to optimal
- Length of a test sequence
  - How close to optimal
- Time to generate a test sequence
  - How fast generation

Experiments and Evaluation (cont.)

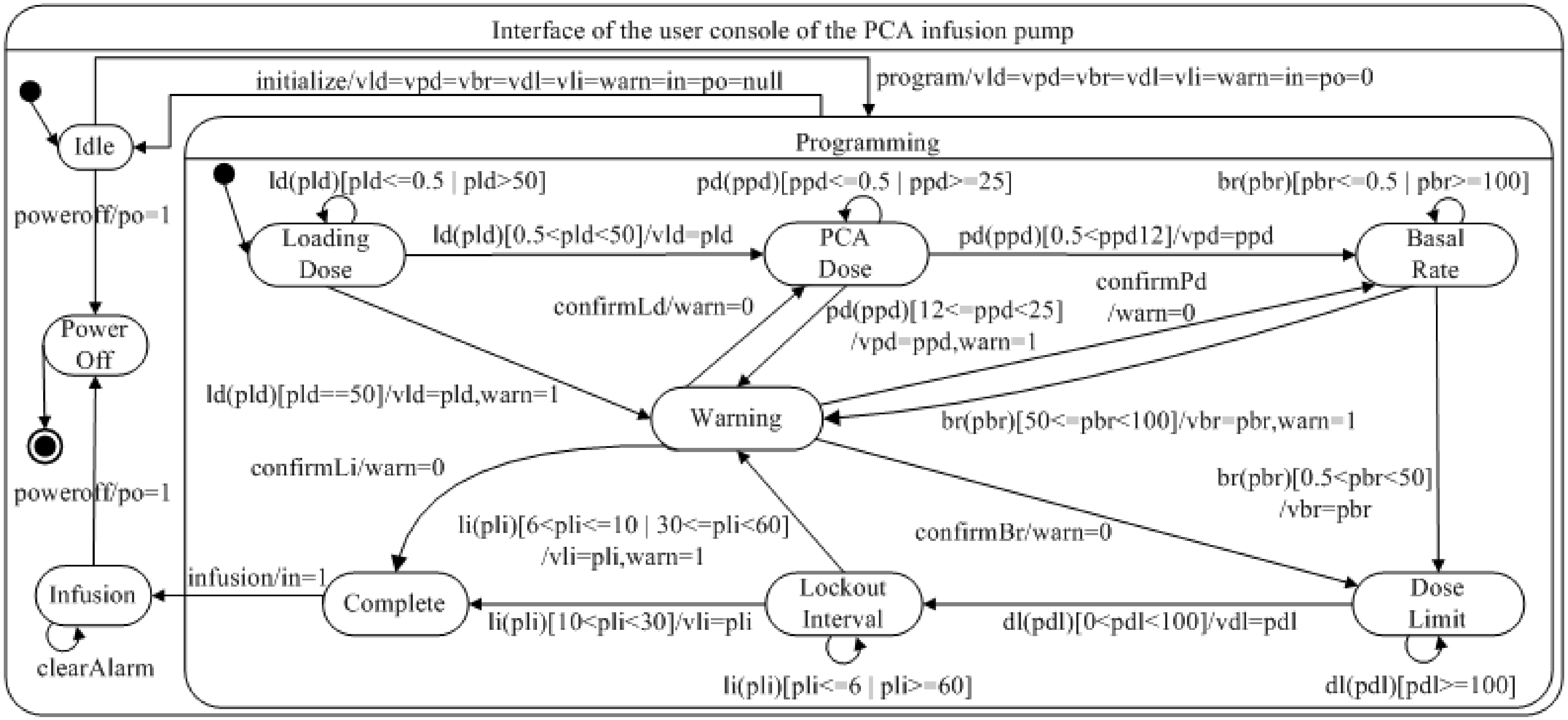
- The length of the generated test sequence
  - compared with the lower bound length
    - branch-and-bound algorithm for finding optimal paths
    - ideal and may not be feasible to execute
  - no shorter test sequence than the lower bound



$$MinBound = Observ(initial\ state) + MinDist(initial\ state,\ E_c) + \sum_i weight(e_c^i) + \boxed{\sum_i MinDist(target(e_c^i), E_c - e_c^i)}$$
$$- Max(\{MinDist(target(e_c^i), E_c - e_c^i) \mid 0 \le i \le n-1\})$$

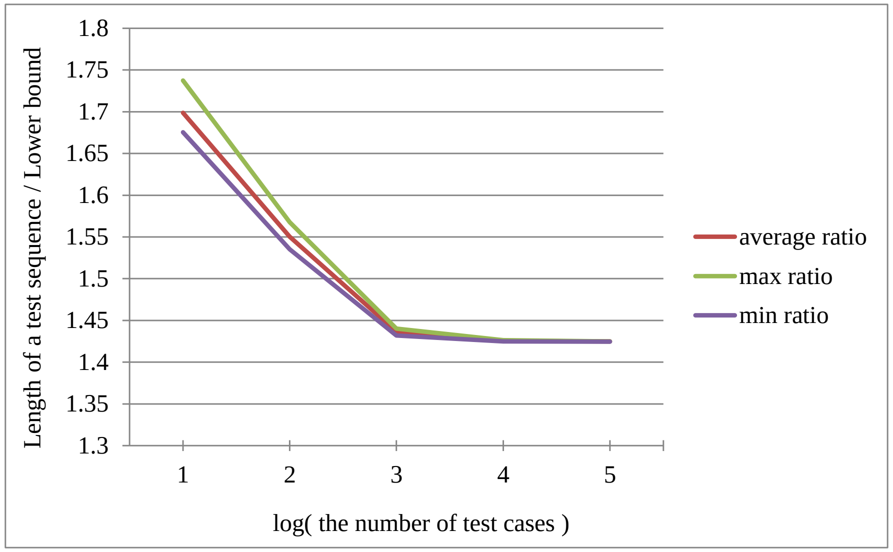
Experiments and Evaluation (cont.)

- PCA Infusion Pump

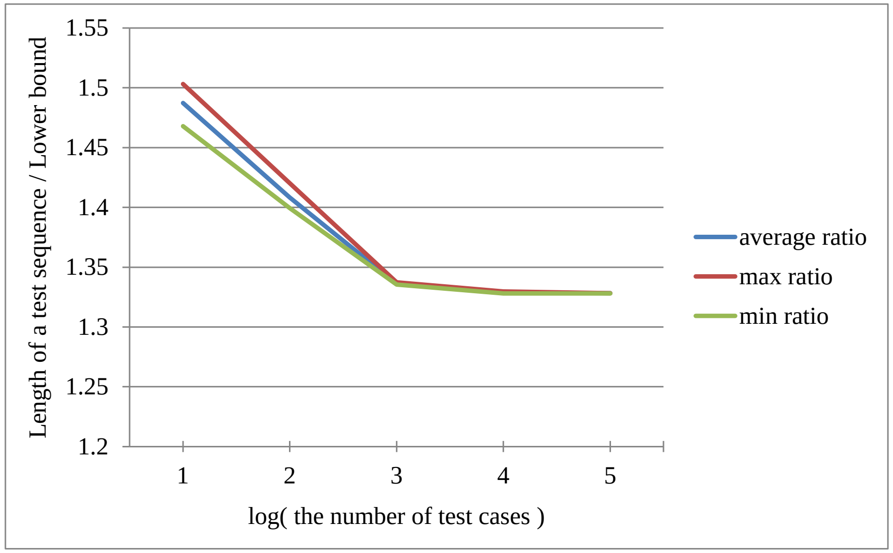

Interface of the user console of the PCA infusion pump
initialize/vld=vpd=vbr=vdl=vli=warn=in=po=null
program/vld=vpd=vbr=vdl=vli=warn=in=po=0
Idle
poweroff/po=1
Power Off
poweroff/po=1
Infusion
clearAlarm
infusion/in=1
Programming
ld(pld)[pld<=0.5 | pld>50]
Loading Dose
ld(pld)[0.5<pld<50]/vld=pld
pd(ppd)[ppd<=0.5 | ppd>=25]
PCA Dose
pd(ppd)[0.5<ppd12]/vpd=ppd
br(pbr)[pbr<=0.5 | pbr>=100]
Basal Rate
confirmLd/warn=0
pd(ppd)[12<=ppd<25] /vpd=ppd,warn=1
confirmPd /warn=0
ld(pld)[pld==50]/vld=pld,warn=1
Warning
br(pbr)[50<=pbr<100]/vbr=pbr,warn=1
confirmLi/warn=0
br(pbr)[0.5<pbr<50] /vbr=pbr
li(pli)[6<pli<=10 | 30<=pli<60] /vli=pli,warn=1
confirmBr/warn=0
Complete
li(pli)[10<pli<30]/vli=pli
Lockout Interval
dl(pdl)[0<pdl<100]/vdl=pdl
Dose Limit
li(pli)[pli<=6 | pli>=60]
dl(pdl)[pdl>=100]

Experiments and Evaluation (cont.)

- PCA Infusion Pump: The length of the generated sequence
  - CPU: Intel Core2 Quad Q6600
  - Main Memory: 8GB
  - Ubuntu 10.10



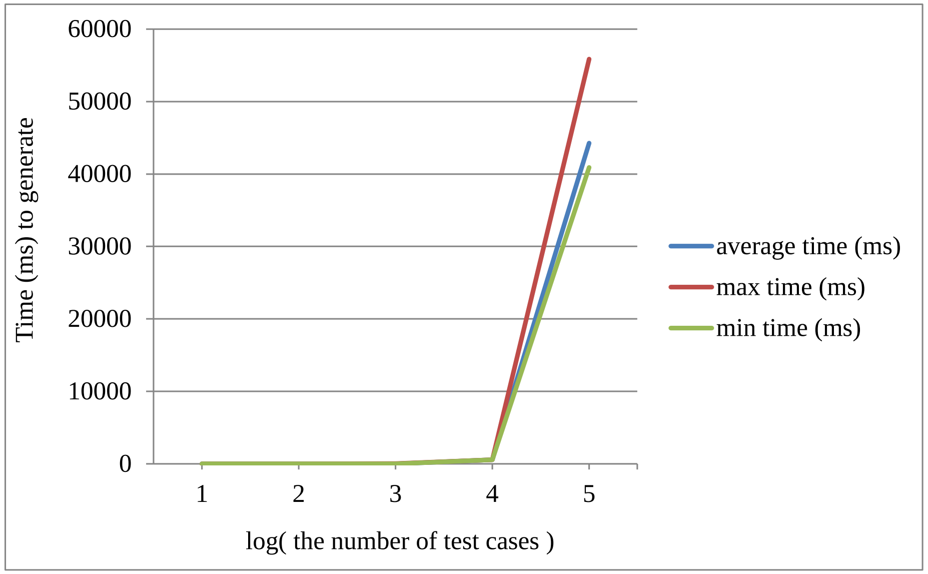
Full Edge Coverage

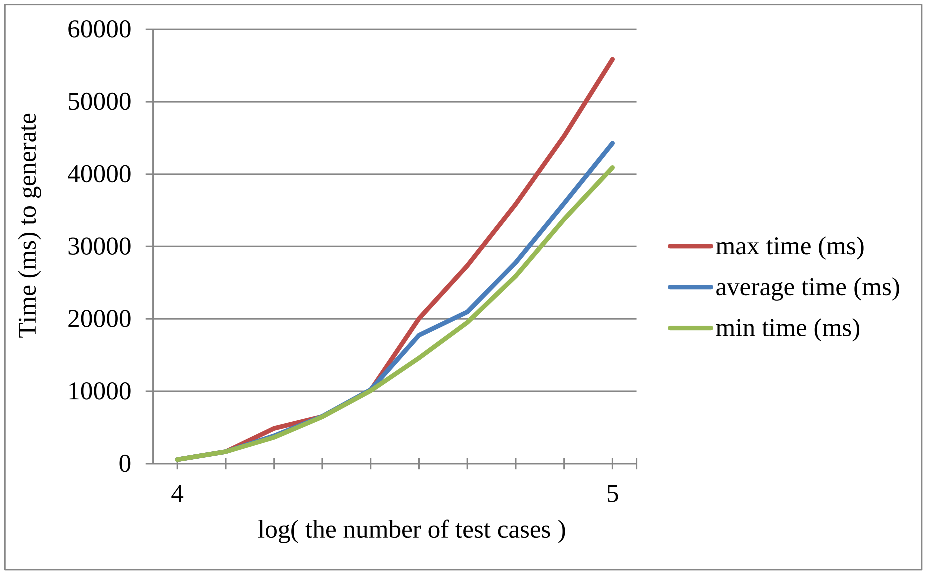


Specific Edge Coverage

Experiments and Evaluation (cont.)

- PCA Infusion Pump: Time to generate the sequence
  - CPU: Intel Core2 Quad Q6600
  - Main Memory: 8GB
  - Ubuntu 10.10

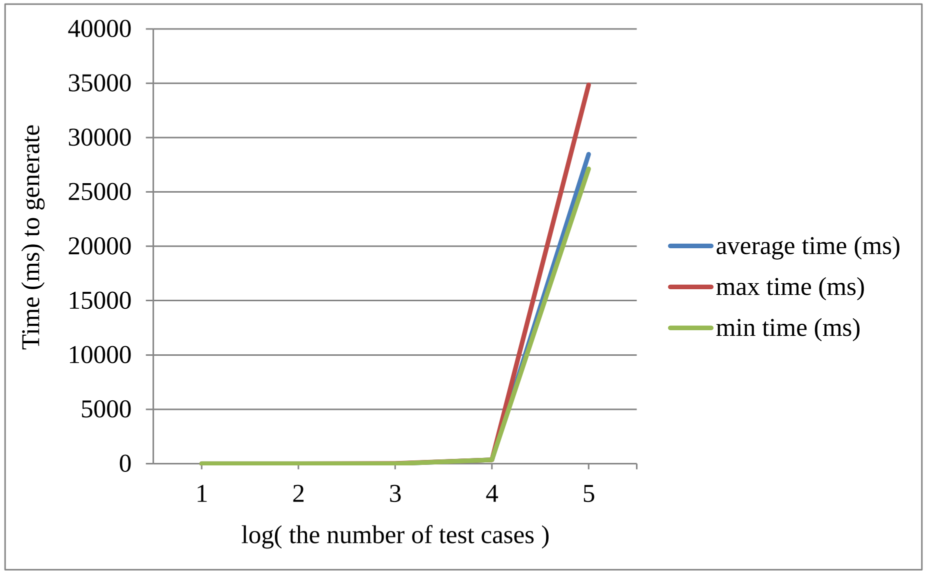


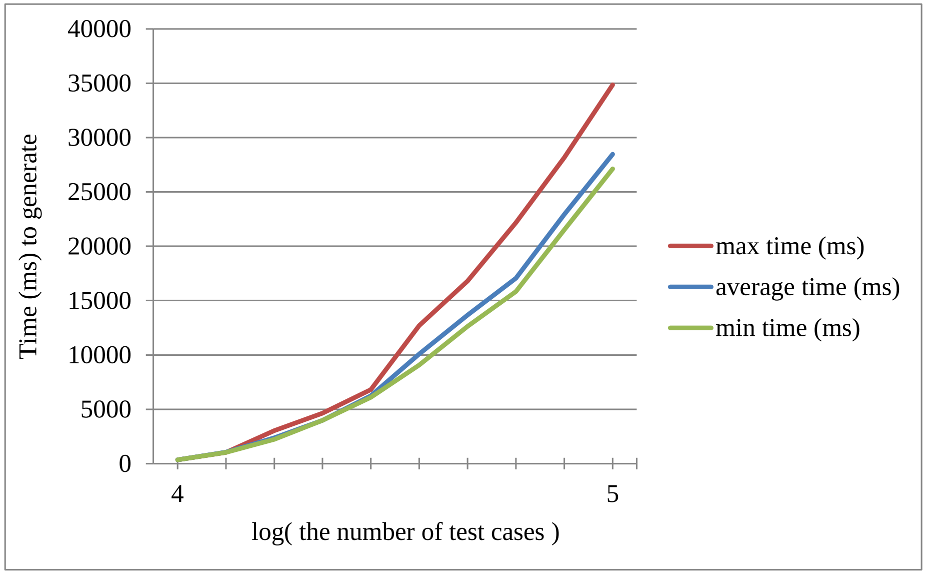


Full Edge Coverage

Experiments and Evaluation (cont.)

- PCA Infusion Pump: Time to generate the sequence
  - CPU: Intel Core2 Quad Q6600
  - Main Memory: 8GB
  - Ubuntu 10.10

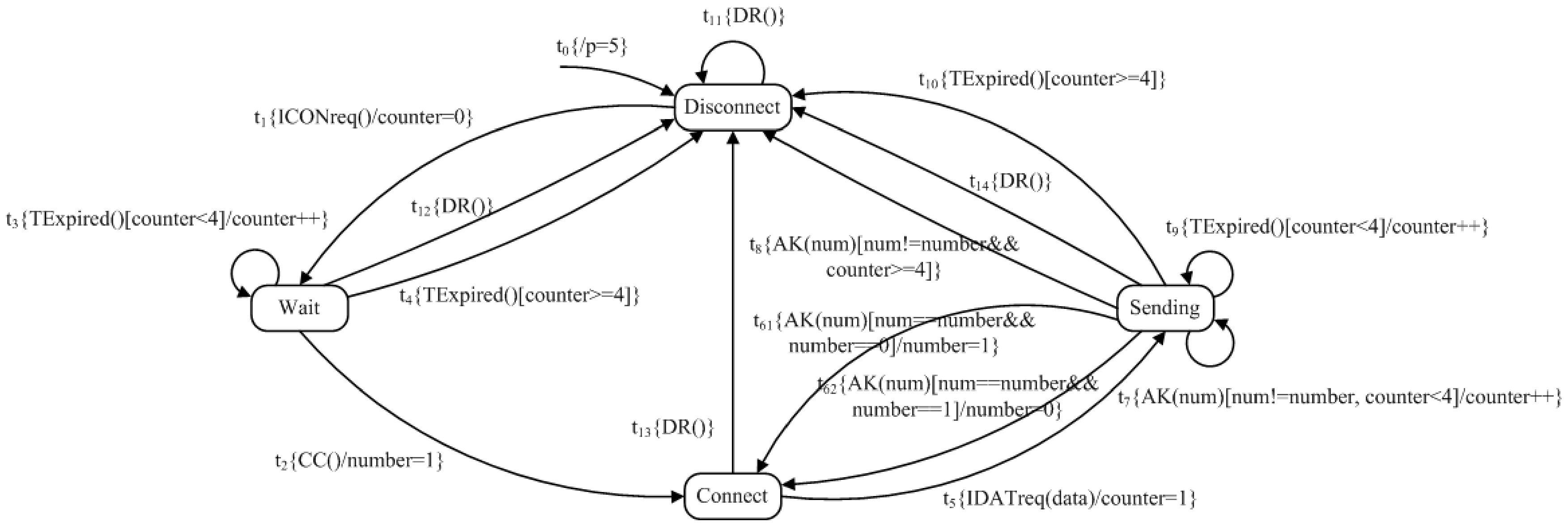


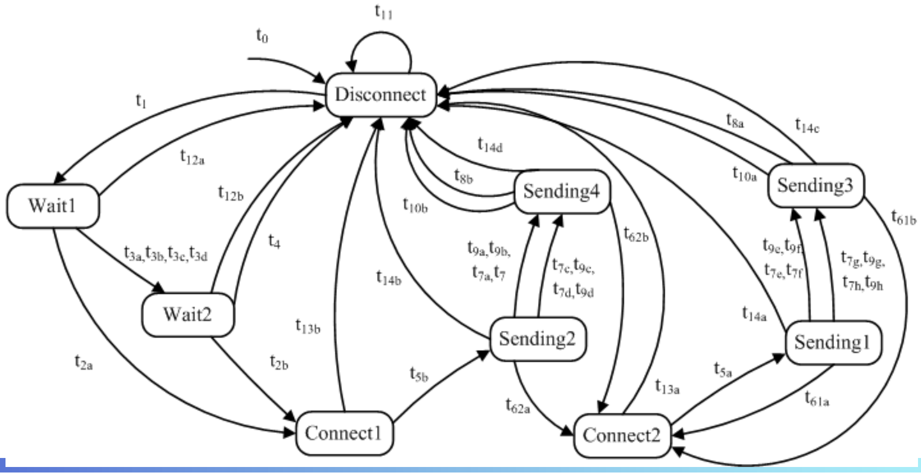


Specific Edge Coverage

Experiments and Evaluation (cont.)

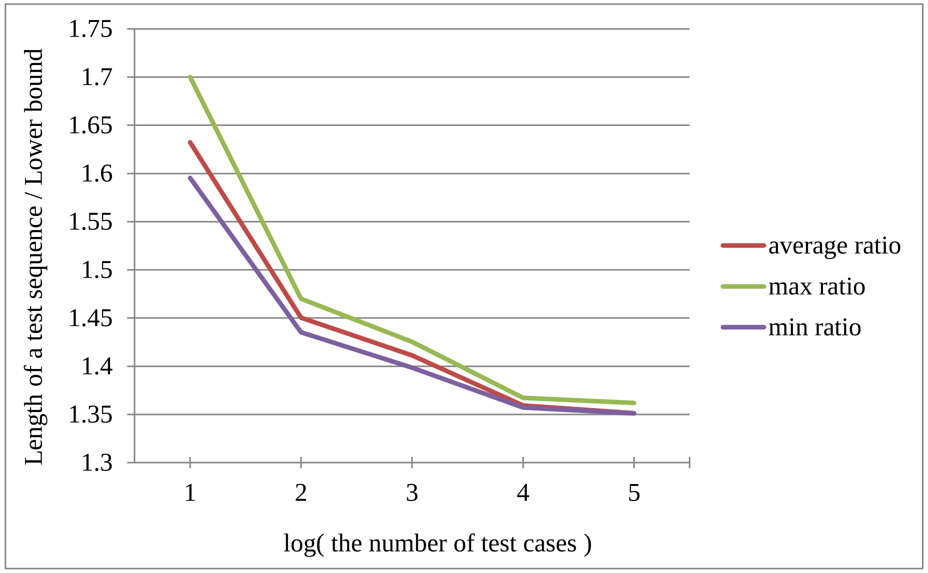
- *Initiator* process of *Inres* Protocol


t₁₁{DR()}
t₀{/p=5}
Disconnect
t₁₀{TExpired()[counter>=4]}
t₁{ICONreq()/counter=0}
t₁₄{DR()}
t₁₂{DR()}
t₃{TExpired()[counter<4]/counter++}
t₉{TExpired()[counter<4]/counter++}
t₈{AK(num)[num!=number&& counter>=4]}
Wait
t₄{TExpired()[counter>=4]}
Sending
t₆₁{AK(num)[num==number&& number==0]/number=1}
t₆₂{AK(num)[num==number&& number==1]/number=0}
t₇{AK(num)[num!=number, counter<4]/counter++}
t₁₃{DR()}
t₂{CC()/number=1}
Connect
t₅{IDATreq(data)/counter=1}

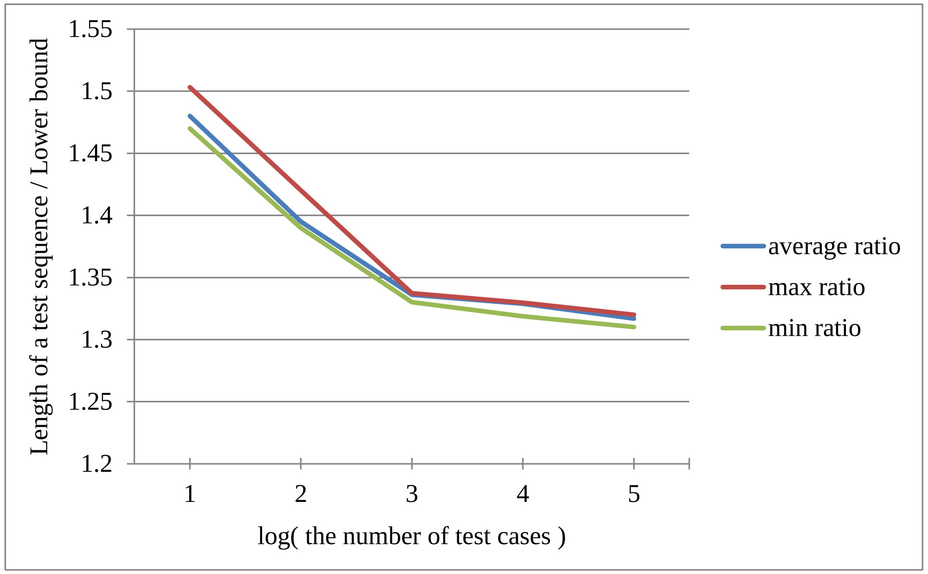


Experiments and Evaluation (cont.)

- *Initiator* Process: The length of the generated sequence
  - CPU: Intel Core2 Quad Q6600
  - Main Memory: 8GB
  - Ubuntu 10.10



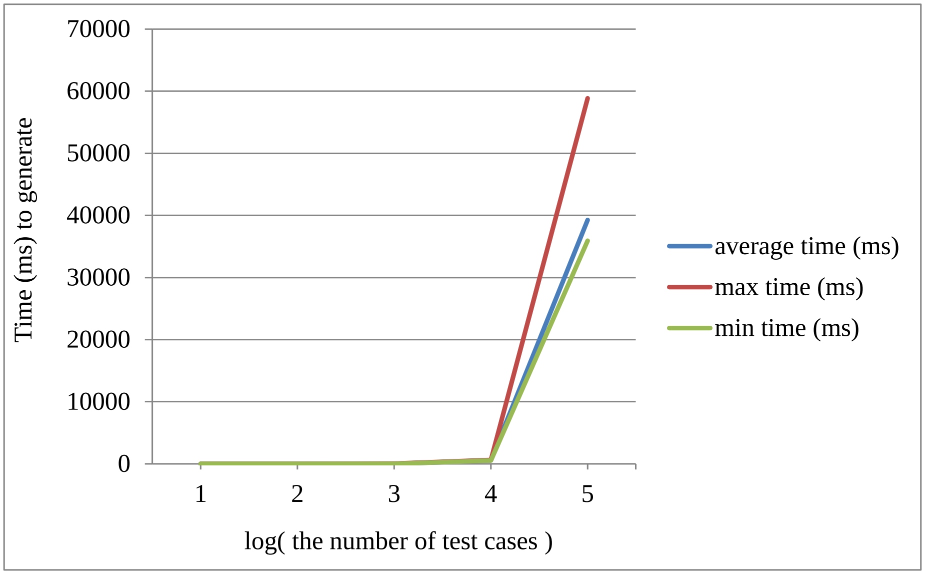
Full Edge Coverage

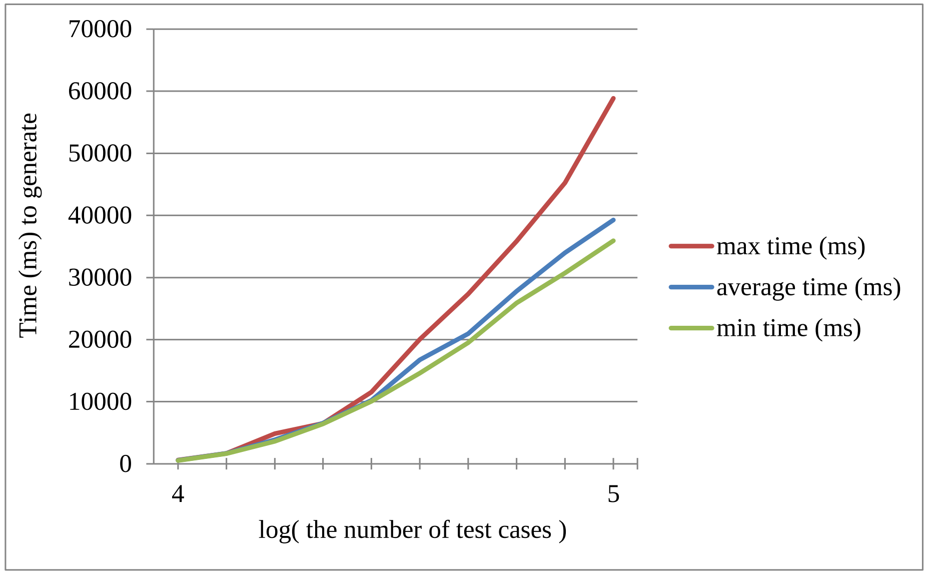


Specific Edge Coverage

Experiments and Evaluation (cont.)

- *Initiator* Process: Time to generate the sequence
  - CPU: Intel Core2 Quad Q6600
  - Main Memory: 8GB
  - Ubuntu 10.10

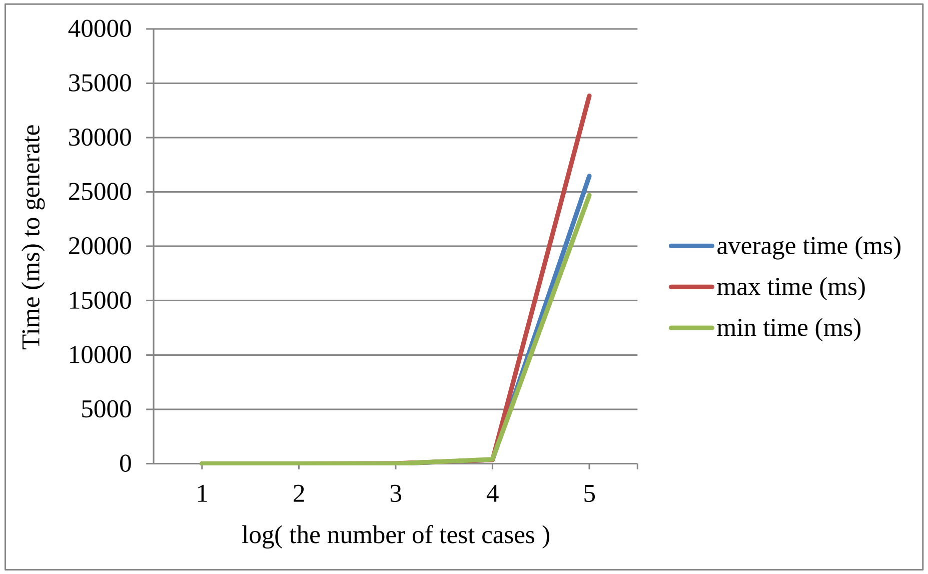


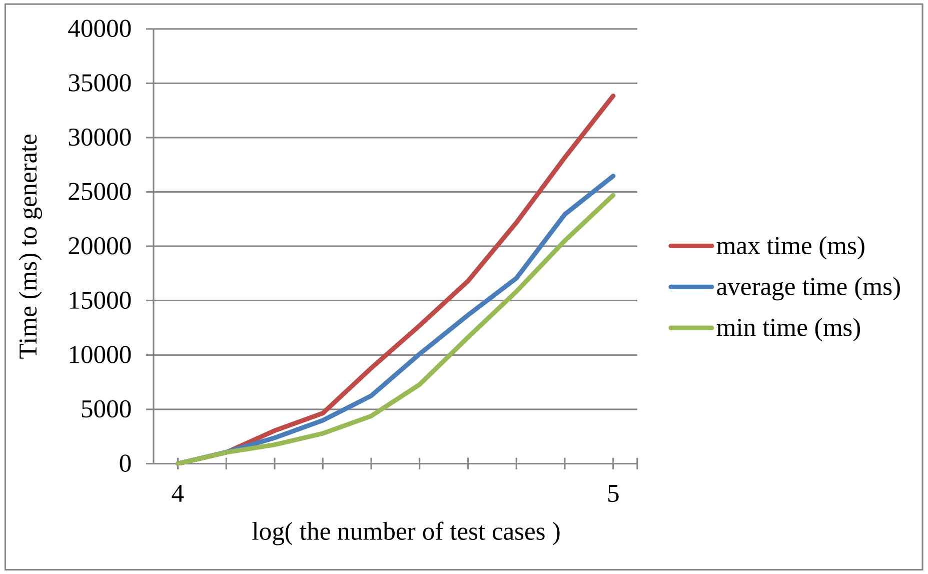


Full Edge Coverage

Experiments and Evaluation (cont.)

- *Initiator* Process : Time to generate the sequence
  - CPU: Intel Core2 Quad Q6600
  - Main Memory: 8GB
  - Ubuntu 10.10





Specific Edge Coverage

Conclusion

- Reusing test cases defined at the unit testing phase
  - Test cases written in source code
  - Mapping the test cases onto interface models
    - **State recognition**
  - Interface testing as high as the unit testing coverage

- Automatic generation of a test sequence
  - **Greedy approach**
    - applies to general models
  - Quickly finds a solution close to optimal

- A tester is given significant test cases
  - that inspect diverse execution paths

- Suitable to complicated software
  - needs plenty of test cases for high-confidence

Conclusion

Thank You

Any Question?