Model-based testing

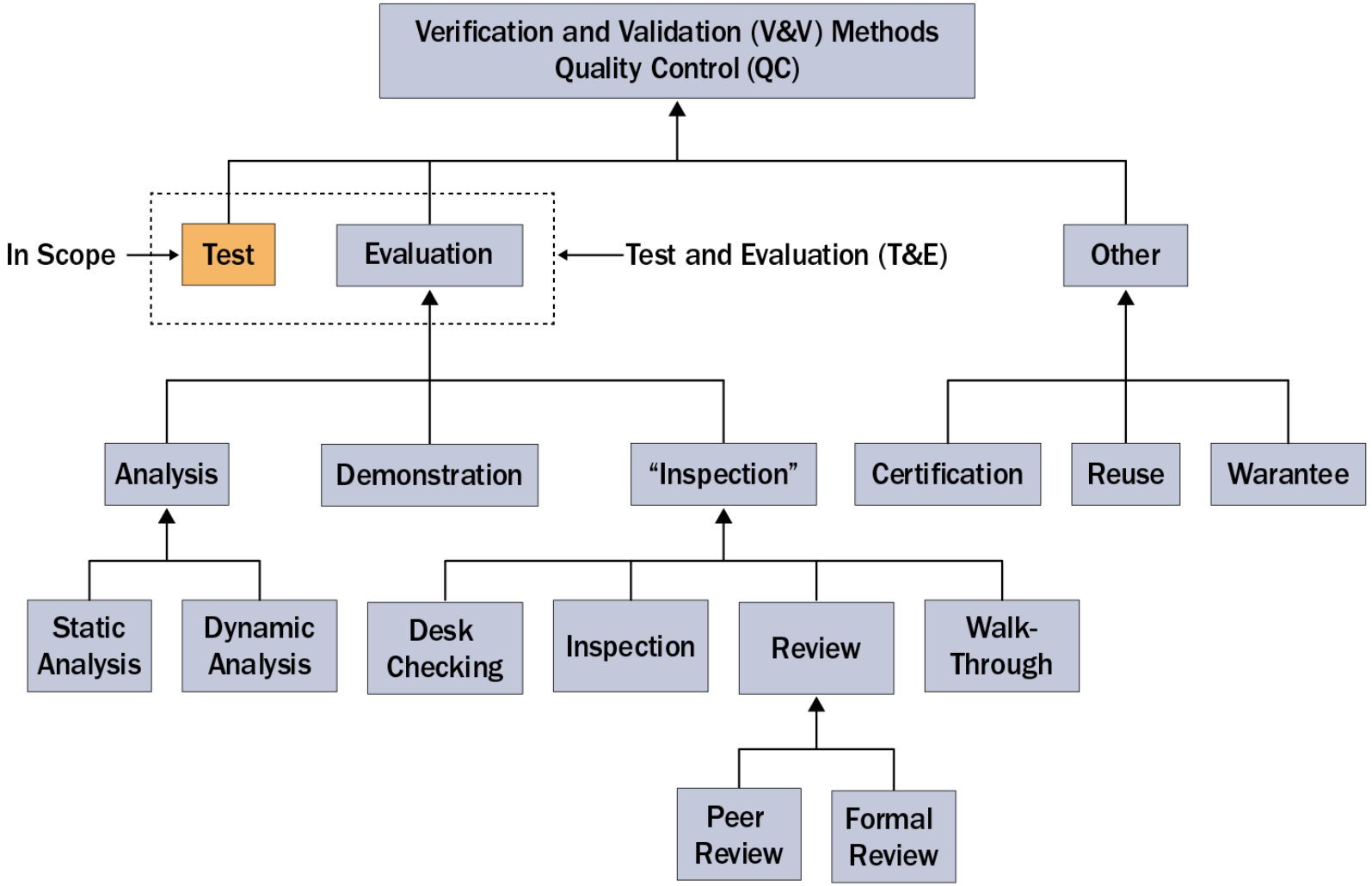
Yunja Choi

Software Safety Engineering Lab. KNU

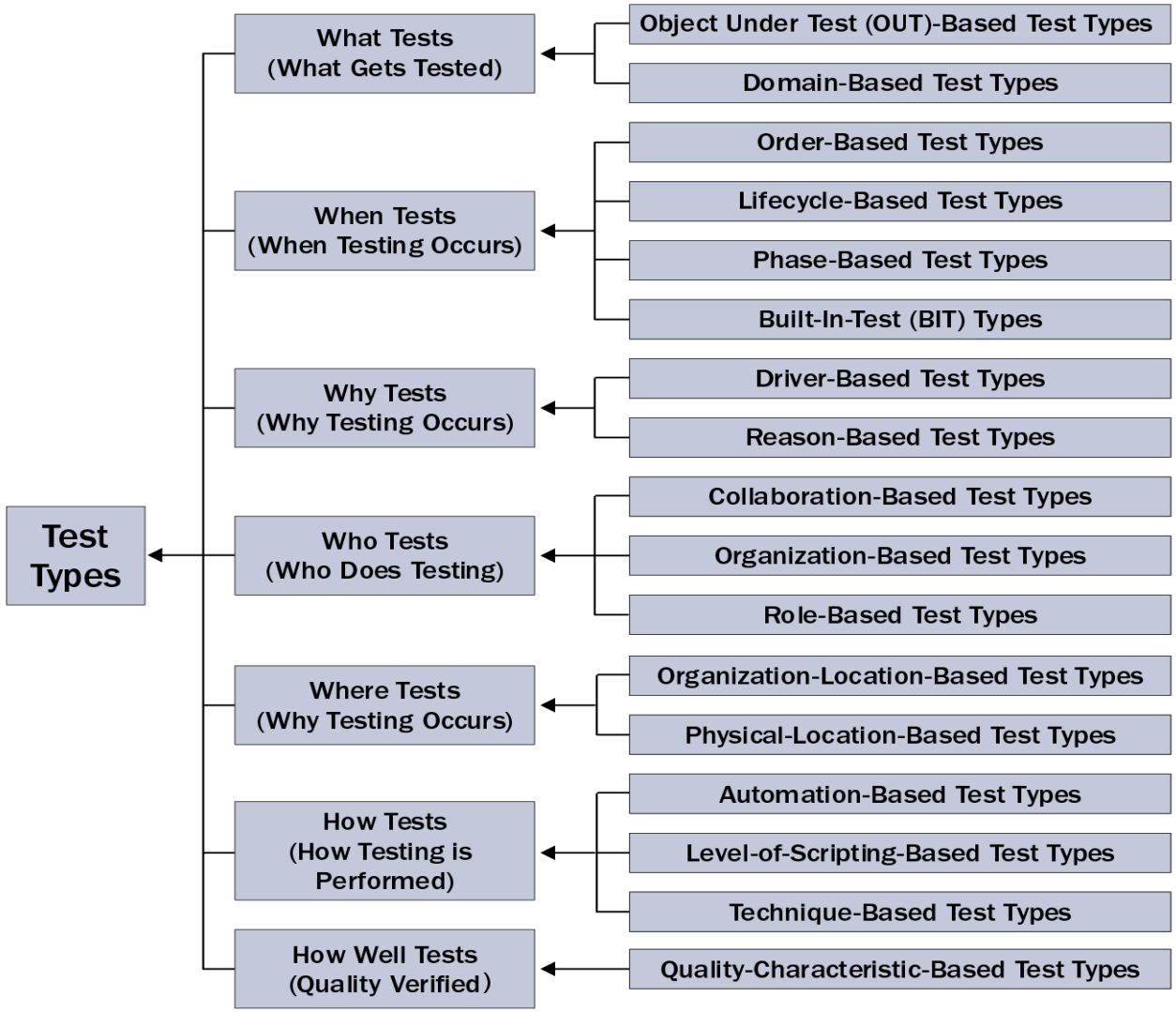
KNU SSELAB

Overview

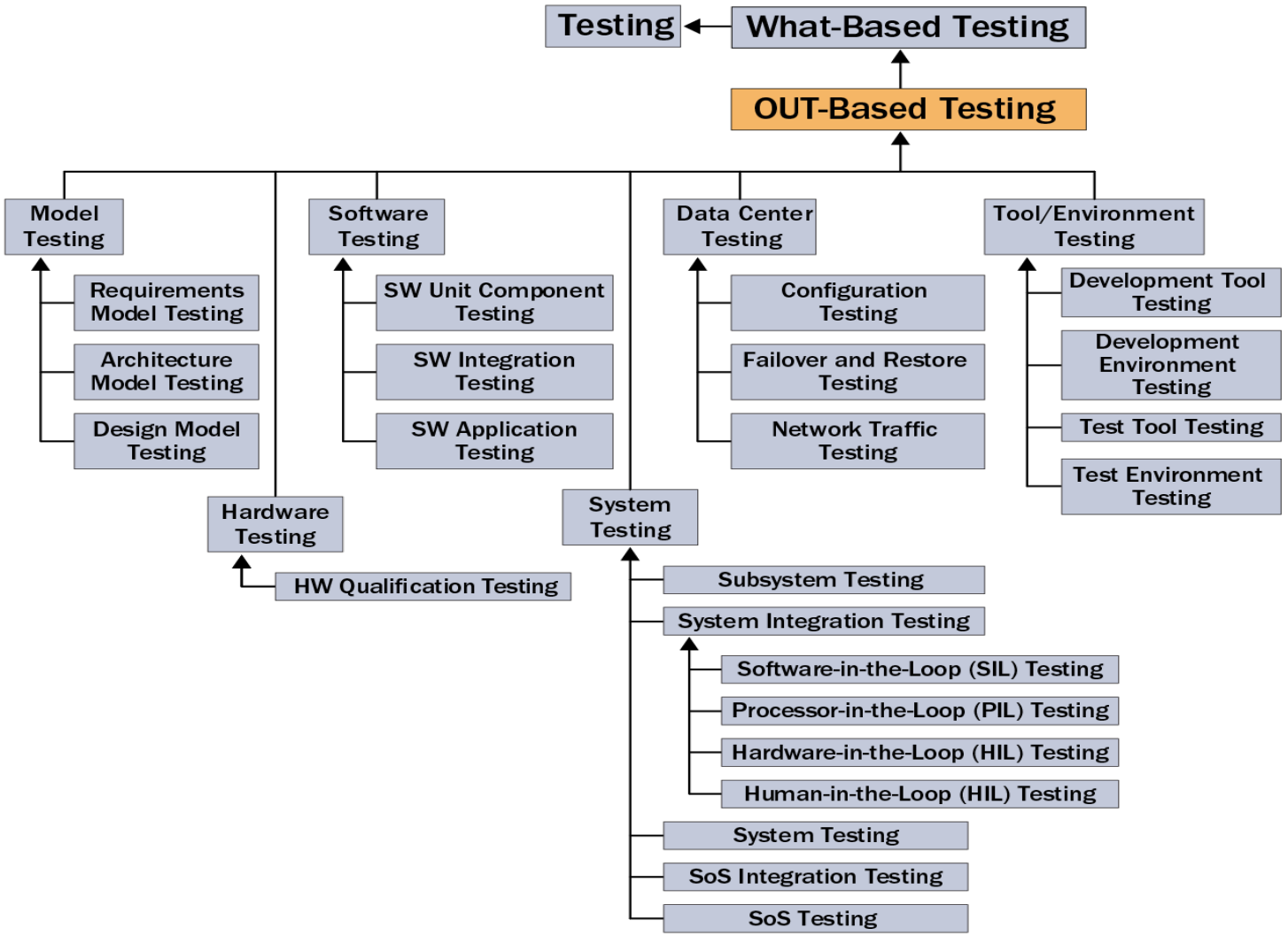
Taxonomy of testing (Donald Firesmith, CMU SEI, 2015)



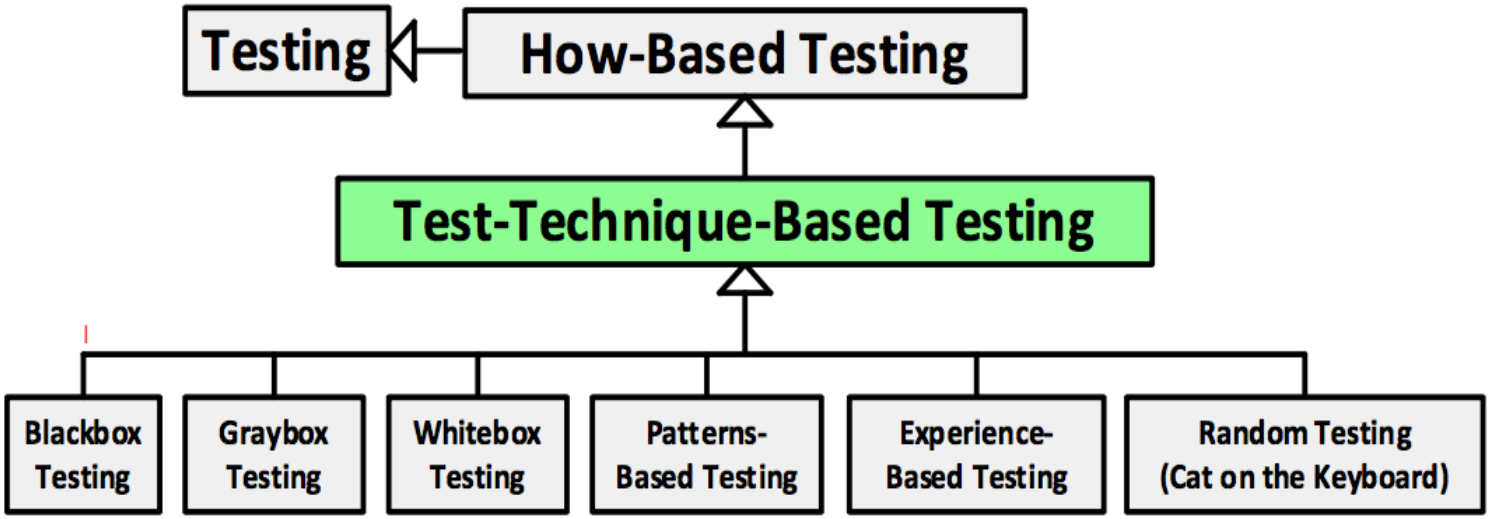
Taxonomy of testing (Donald Firesmith, CMU SEI, 2015)



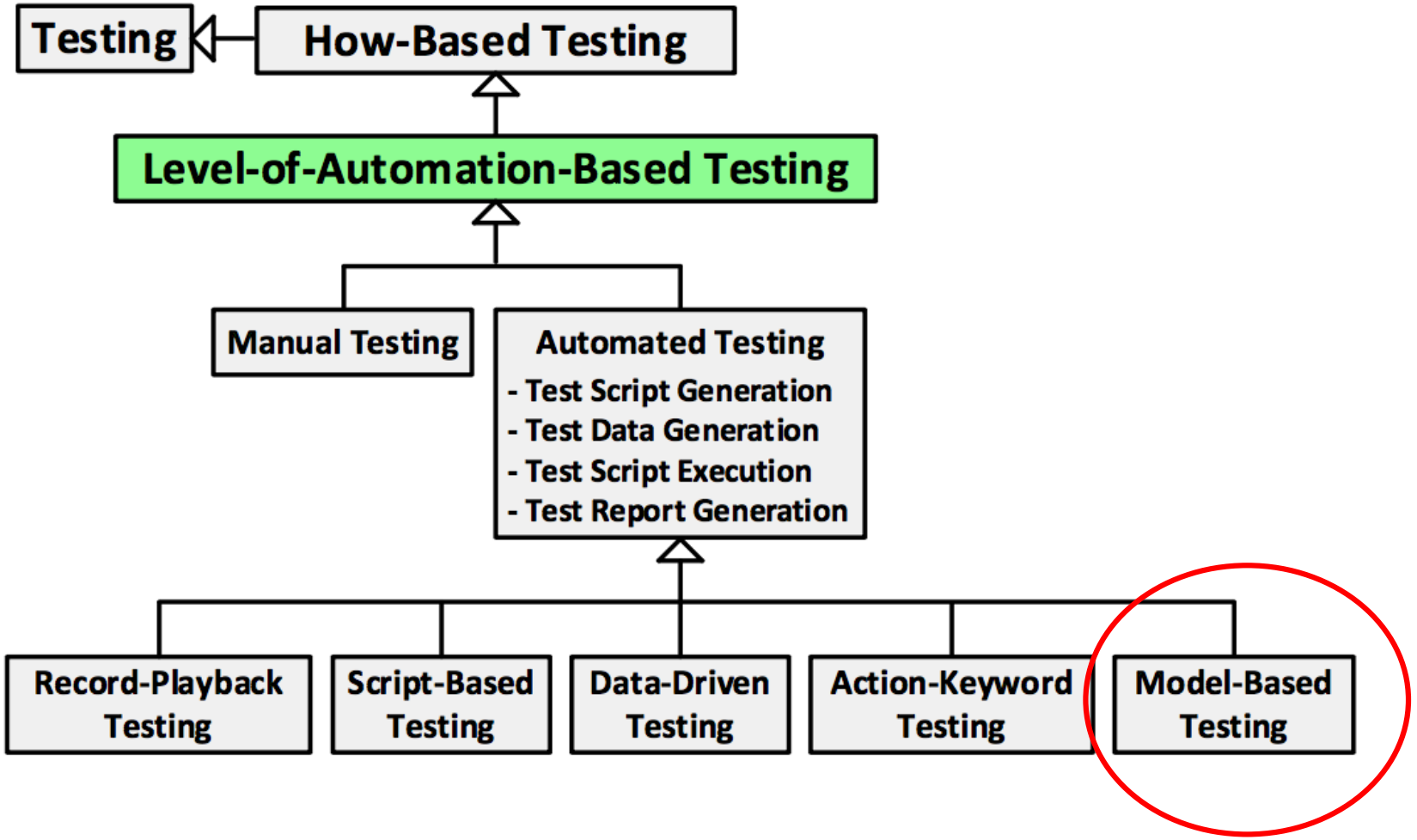
Taxonomy of testing (Donald Firesmith, CMU SEI, 2015)



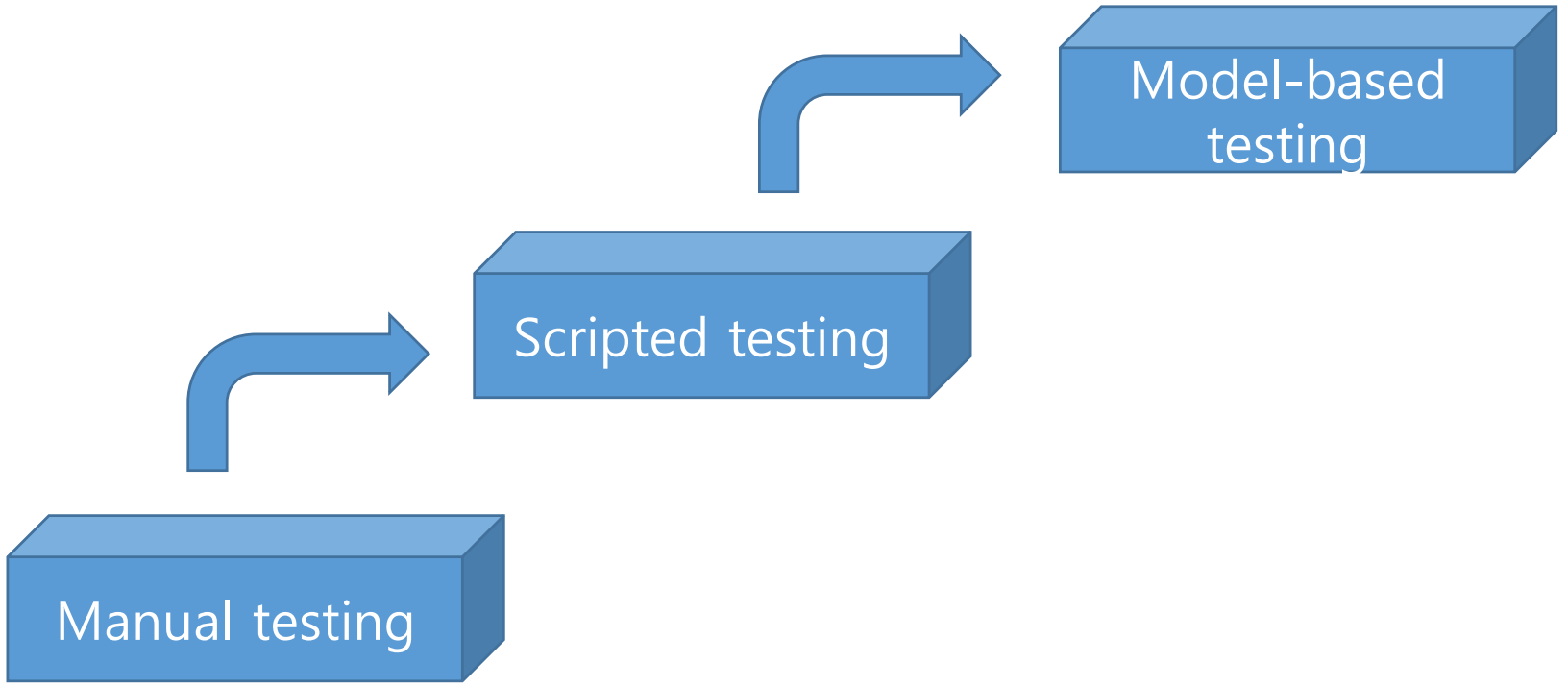
Taxonomy of testing (Donald Firesmith, CMU SEI, 2015)



Taxonomy of testing (Donald Firesmith, CMU SEI, 2015)



Software testing evolution



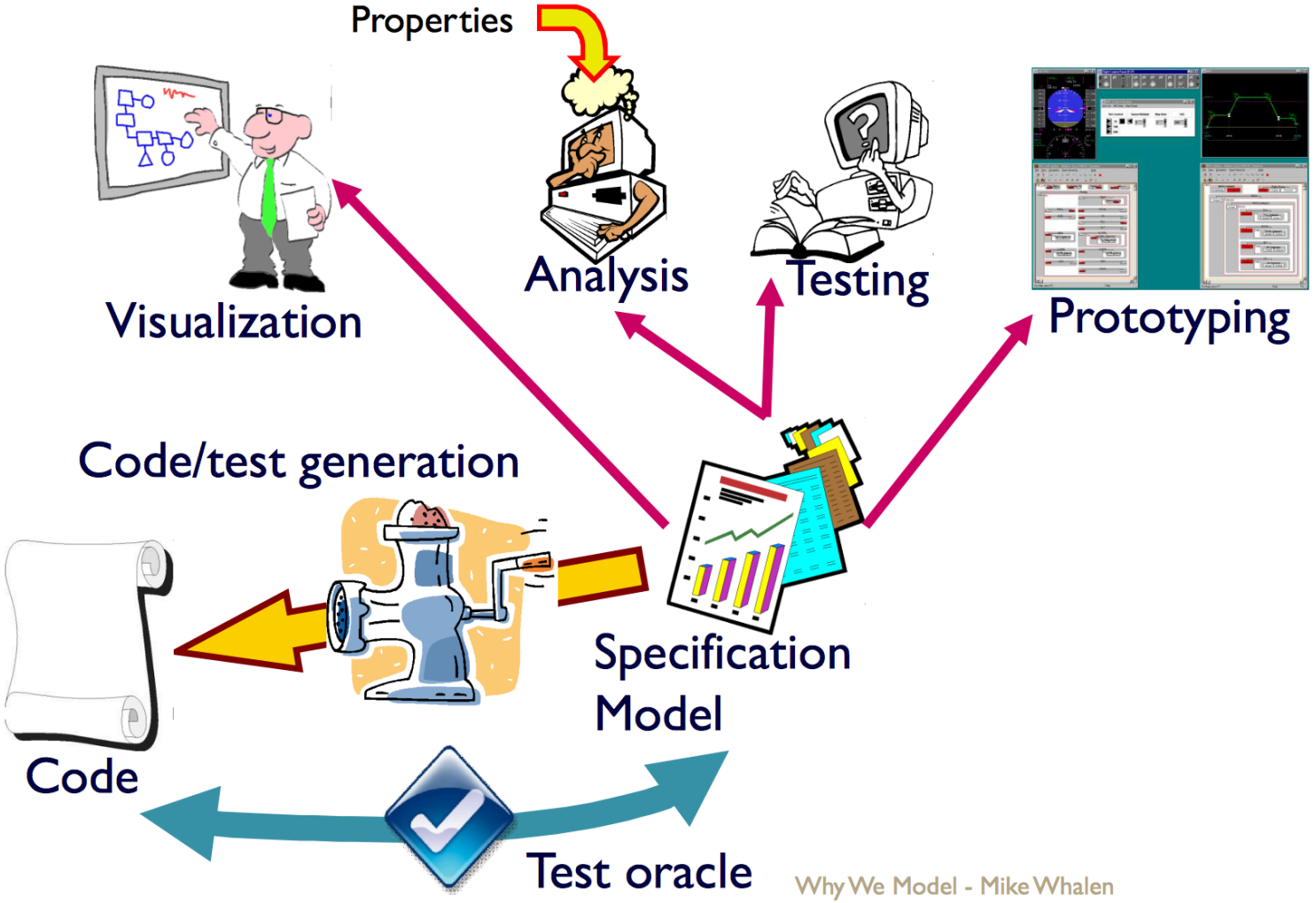
What is a model?

- A representation of a system using general rules and concept s
- A model can come in many shapes, sizes, and styles.
- A model is not the real world, but a human construct to help us better understand the real world.
- All models have an information input, an information process or, and an output of expected results

Why model-based? (development perspective)

- To make sure that we understand what we want to develop
- To make sure that what we develope is what we wanted to develop in the first place (validation)
- Models are easier to understand than natural language documents
  - Validation through simulation is possible
  - Completeness and consistency of the model can be checked automatically
- Development can be entirely model-based through step-by-step refinements
  - Entire development process can be automated using model-based code generation

Model-based development
(Mike Whalen, ICSE2013 Tutorial)



Model-based development examples

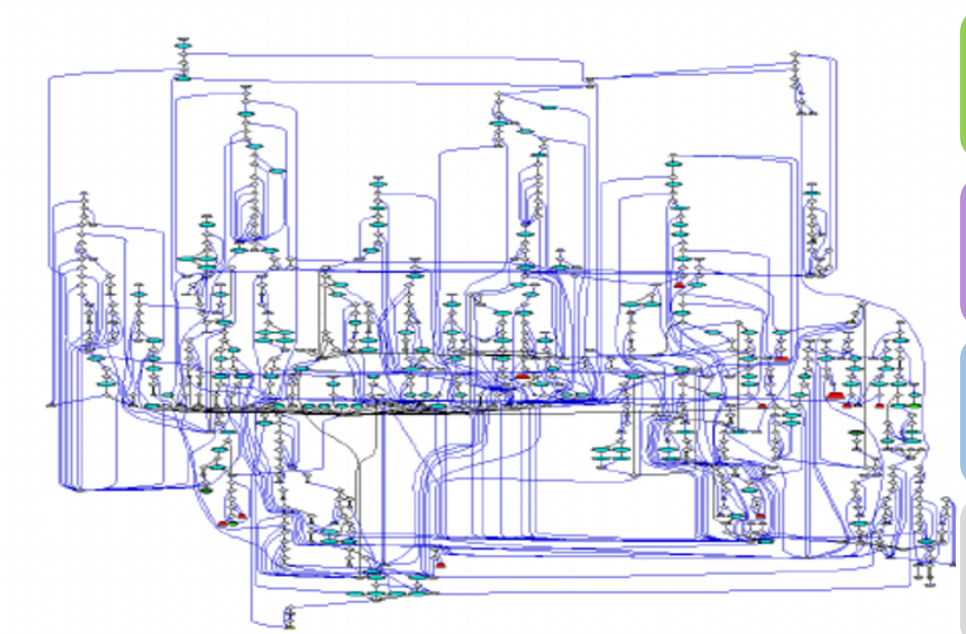
Company	Product	Tools	Specified & Autocoded	Benefits Claimed
Airbus	A340	SCADE With Code Generator	• 70% Fly-by-wire Controls • 70% Automatic Flight Controls • 50% Display Computer • 40% Warning & Maint Computer	• 20X Reduction in Errors • Reduced Time to Market
Eurocopter	EC-155/135 Autopilot	SCADE With Code Generator	• 90 % of Autopilot	• 50% Reduction in Cycle Time
GE & Lockheed Martin	FADEDC Engine Controls	ADI Beacon	• Not Stated	• Reduction in Errors • 50% Reduction in Cycle Time • Decreased Cost
Schneider Electric	Nuclear Power Plant Safety Control	SCADE With Code Generator	• 200,000 SLOC Auto Generated from 1,200 Design Views	• 8X Reduction in Errors while Complexity Increased 4x
US Spaceware	DCX Rocket	MATRIXx	• Not Stated	• 50-75% Reduction in Cost • Reduced Schedule & Risk
PSA	Electrical Management System	SCADE With Code Generator	• 50% SLOC Auto Generated	• 60% Reduction in Cycle Time • 5X Reduction in Errors
CSEE Transport	Subway Signaling System	SCADE With Code Generator	• 80,000 C SLOC Auto Generated	• Improved Productivity from 20 to 300 SLOC/day
Honeywell Commercial Aviation Systems	Primus Epic Flight Control System	MATLAB Simulink	• 60% Automatic Flight Controls	• 5X Increase in Productivity • No Coding Errors • Received FAA Certification

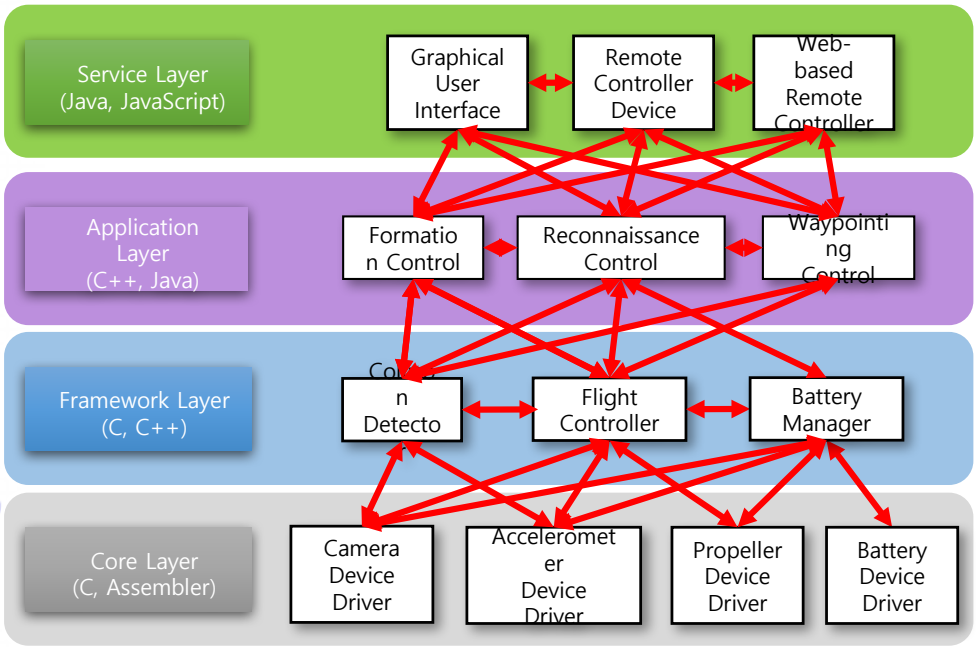
Slide from “Proving the Shalls” by Steve Miller, 2006 Rockwell Collins, Inc. All rights reserved

Why model-based? (testing perspective)

- Manual testing cannot cope with the increasing complexity of software itself
  - Exponential complexity due to
    - The size of software
    - Interactions with users, external systems, and HW
    - The size of inputs

Automation!

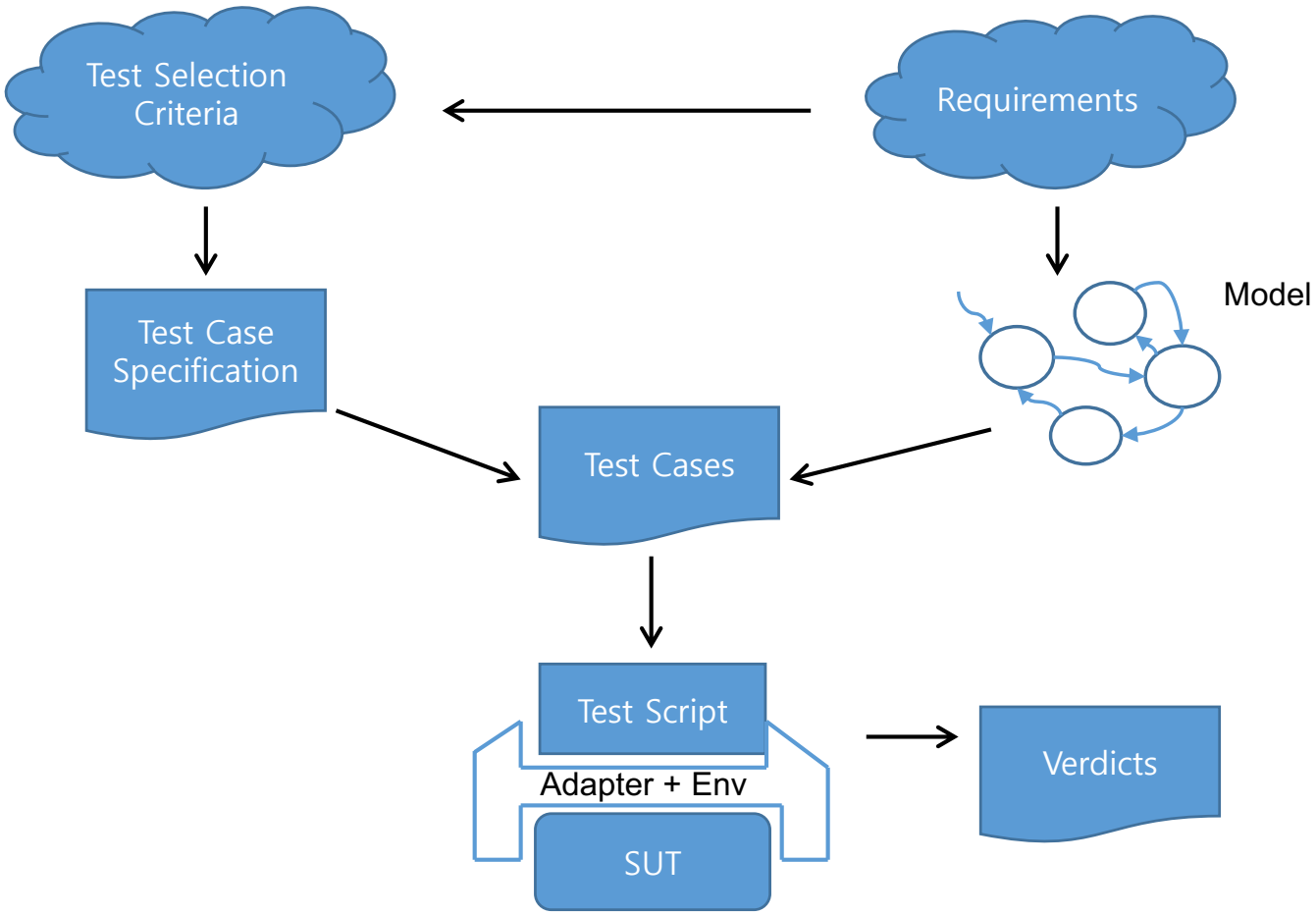




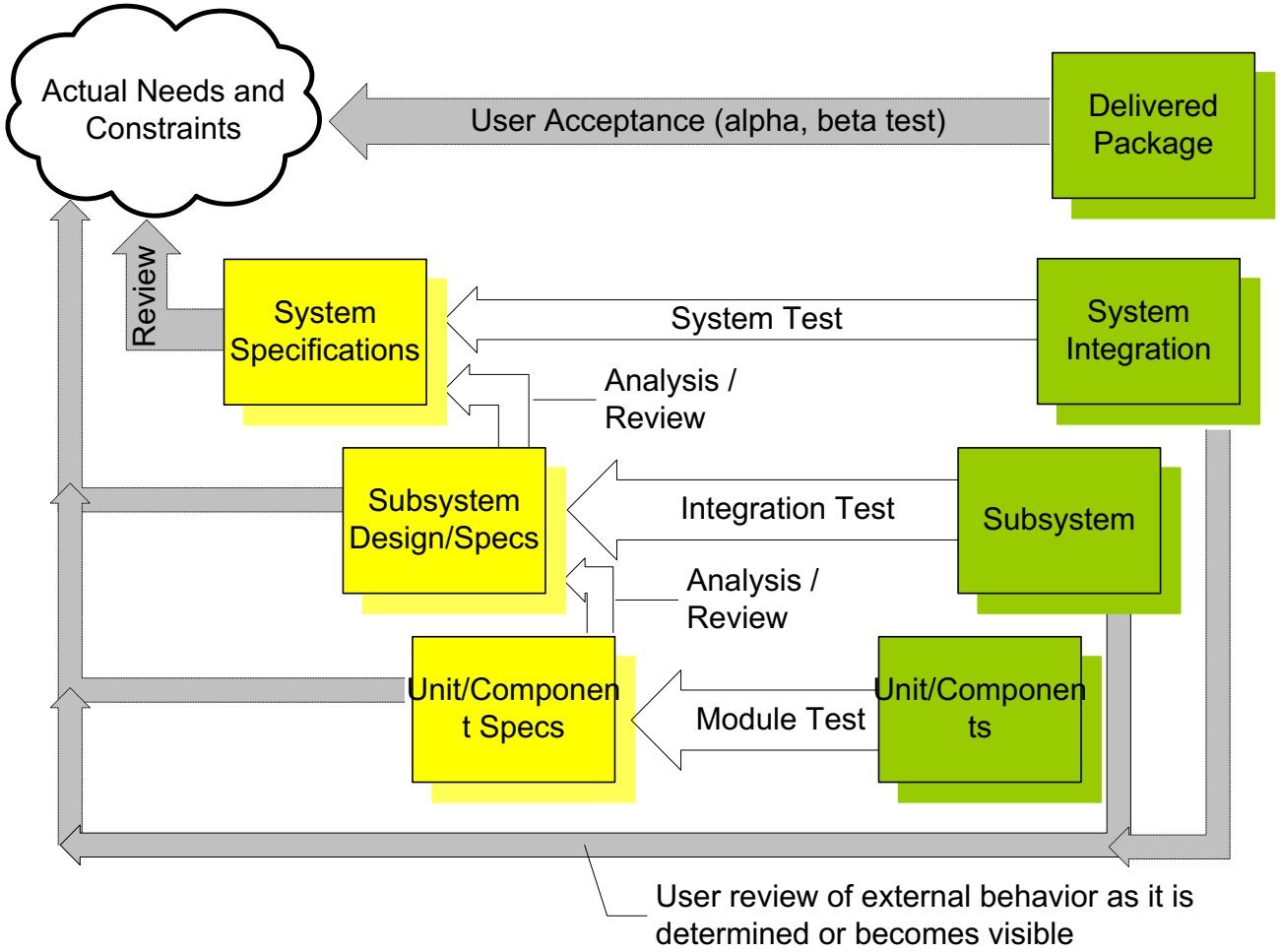
Why model-based? (testing perspective)

- We need a systematic testing ap proach to cope with the growing complexity
- We can measure the quality of t esting using coverage criteria, b ut how to ensure the best cover age?

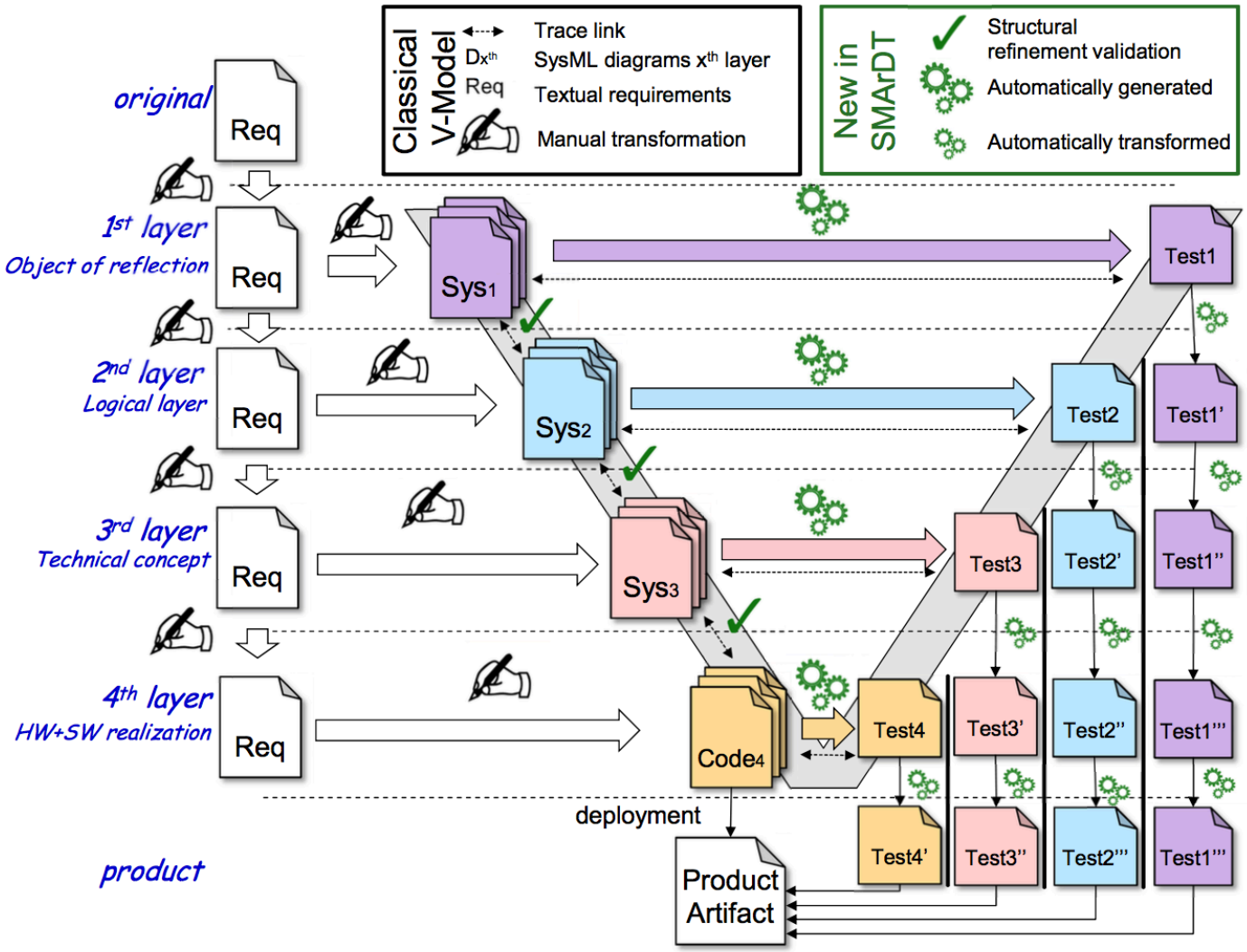
Model-based testing



Determining what to test: V-model



The V model and model-based testing at BMW

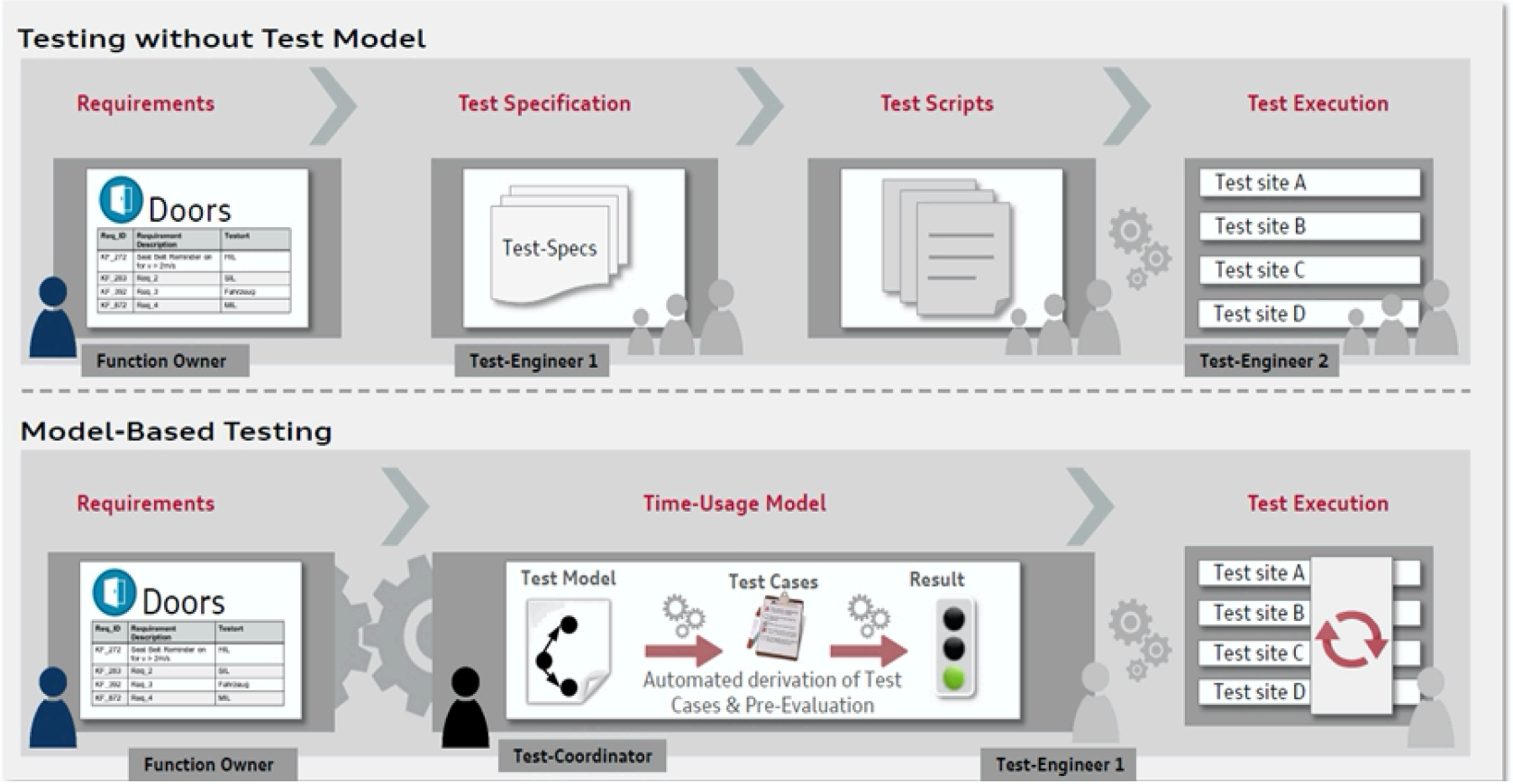


Model-based testing at Audi

- [Audi] ISO26262의 요구사항분석, Test Case(T/C) 개발, **Test** 수행으로 이어지는 기존 방법은 한계 → **MBT** 필요!!

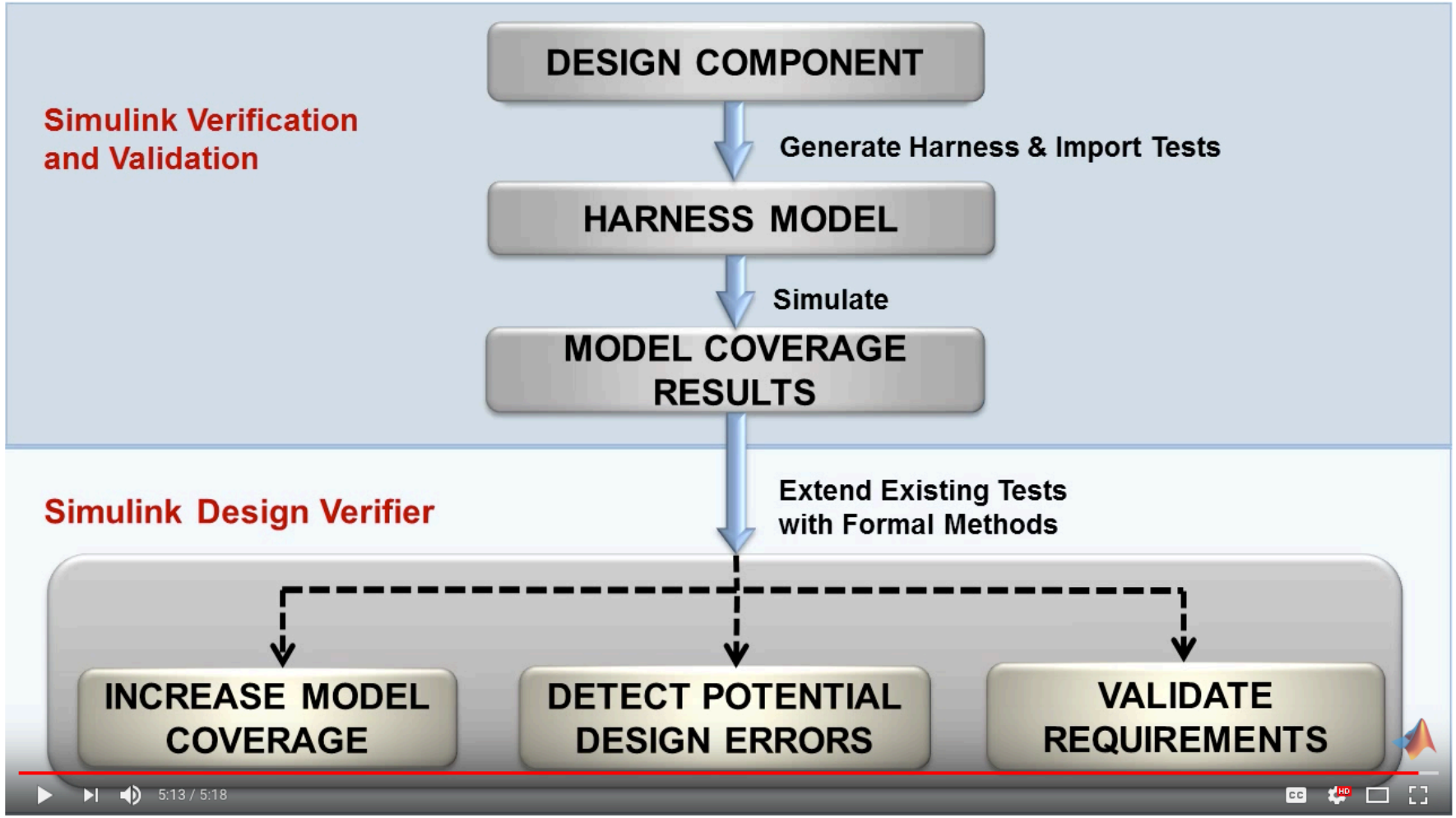
Model Based Testing of Driving Functions at AUDI AG

Comparison of Test processes without and with Test Model



Model-based testing in Simulink

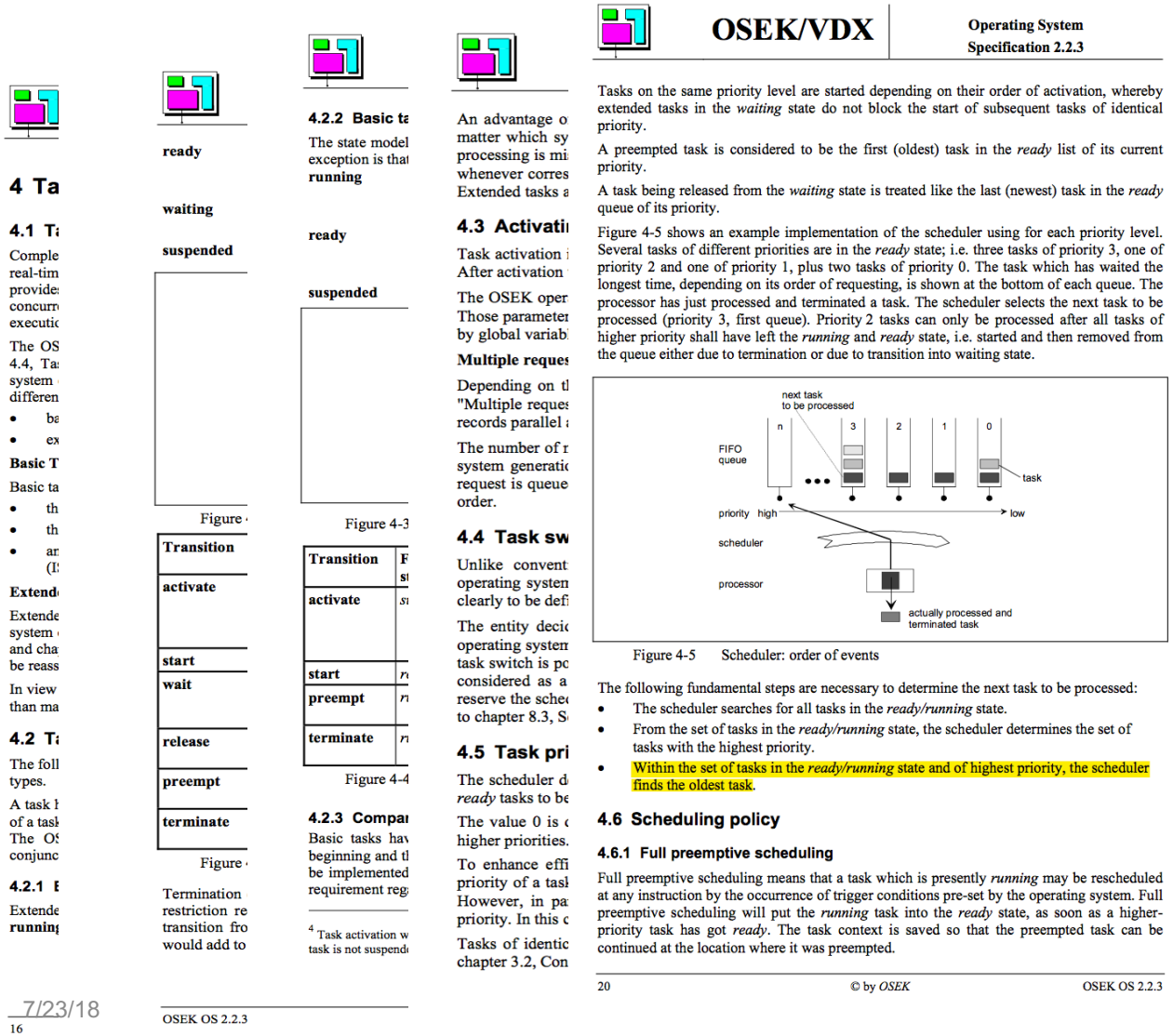
https://www.youtube.com/watch?v=2bO72KEWLNY



Reducing ambiguity & complexity

- Ambiguity
  - Formal model <<< Code < Informal model <<< Requirements
- Complexity
  - Model << Requirements <<< Code

Complexity and ambiguity of requirements



4 Ta

4.1 T

Comple
real-tim
provide
concurr
executi

The OS
4.4, Ta
system
differen

- ba
- ex

Basic T

Basic ta

- th
- th
- ar

Extend

Extende
system
and cha
be reass

In view
than ma

4.2 Ta

The foll
types.

A task
of a task
The OS
conjunc

4.2.1

Extende
running

ready

waiting

suspended

Figure

Transition	
activate	
start	
wait	
release	
preempt	
terminate	

Figure

Termination
restriction re
transition fro
would add to

4.2.2 Basic ta

The state model
exception is that

running

ready

suspended

Figure 4-3

Transition	F
activate	
start	
preempt	
terminate	

Figure 4-4

4.2.3 Compar

Basic tasks hav
beginning and t
be implemented
requirement reg

[4] Task activation w
task is not suspend

An advantage o
matter which sy
processing is mi
whenever corres
Extended tasks a

4.3 Activati

Task activation
After activation

The OSEK oper
Those parameter
by global variab

Multiple reques

Depending on t
"Multiple reques
records parallel

The number of r
system generatio
request is queue
order.

4.4 Task sw

Unlike convent
operating systen
clearly to be defi

The entity decid
operating systen
task switch is po
considered as a
reserve the sched
to chapter 8.3, S

4.5 Task pri

The scheduler d
ready tasks to be

The value 0 is
higher priorities.

To enhance effi
priority of a task
However, in pa
priority. In this c

Tasks of identic
chapter 3.2, Con

OSEK OS 2.2.3 © by *OSEK* 19

OSEK/VDX Operating System Specification 2.2.3

Tasks on the same priority level are started depending on their order of activation, whereby extended tasks in the *waiting* state do not block the start of subsequent tasks of identical priority.

A preempted task is considered to be the first (oldest) task in the *ready* list of its current priority.

A task being released from the *waiting* state is treated like the last (newest) task in the *ready* queue of its priority.

Figure 4-5 shows an example implementation of the scheduler using for each priority level. Several tasks of different priorities are in the *ready* state; i.e. three tasks of priority 3, one of priority 2 and one of priority 1, plus two tasks of priority 0. The task which has waited the longest time, depending on its order of requesting, is shown at the bottom of each queue. The processor has just processed and terminated a task. The scheduler selects the next task to be processed (priority 3, first queue). Priority 2 tasks can only be processed after all tasks of higher priority shall have left the *running* and *ready* state, i.e. started and then removed from the queue either due to termination or due to transition into waiting state.

Figure 4-5 Scheduler: order of events

The following fundamental steps are necessary to determine the next task to be processed:

- The scheduler searches for all tasks in the *ready/running* state.
- From the set of tasks in the *ready/running* state, the scheduler determines the set of tasks with the highest priority.
- Within the set of tasks in the *ready/running* state and of highest priority, the scheduler finds the oldest task.

4.6 Scheduling policy

4.6.1 Full preemptive scheduling

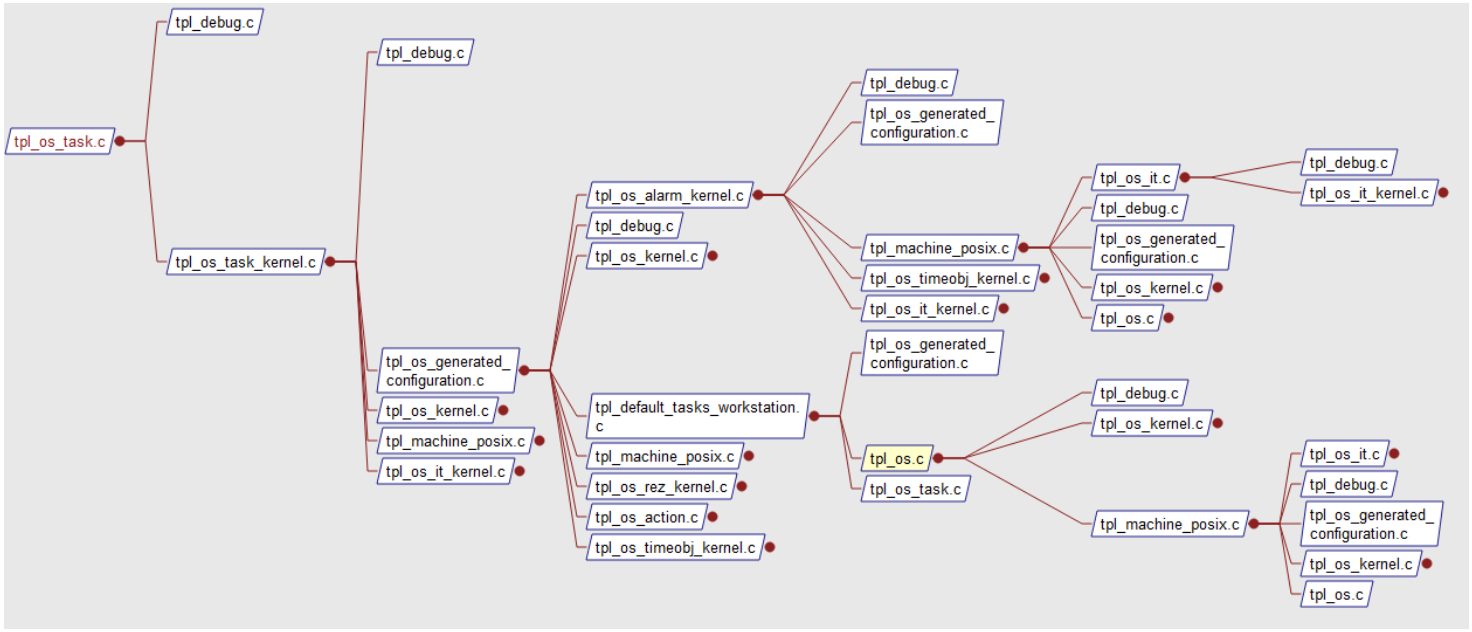
Full preemptive scheduling means that a task which is presently *running* may be rescheduled at any instruction by the occurrence of trigger conditions pre-set by the operating system. Full preemptive scheduling will put the *running* task into the *ready* state, as soon as a higher-priority task has got *ready*. The task context is saved so that the preempted task can be continued at the location where it was preempted.

20 © by *OSEK* OSEK OS 2.2.3

- Complete?
- Consistent?
- Correct?

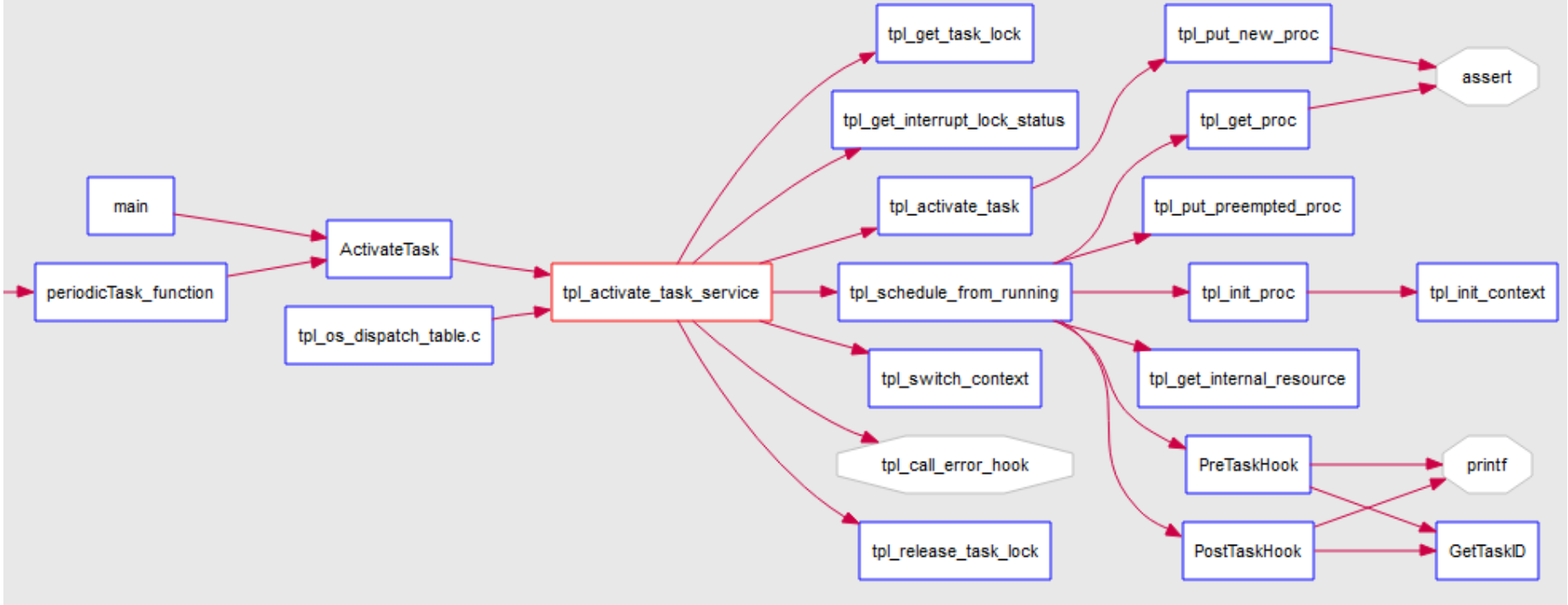
Complexity of code

- About 2100 lines of code only from tpl_os_task.c, tpl_os_task_kernel.c, tpl_os_kernel.c

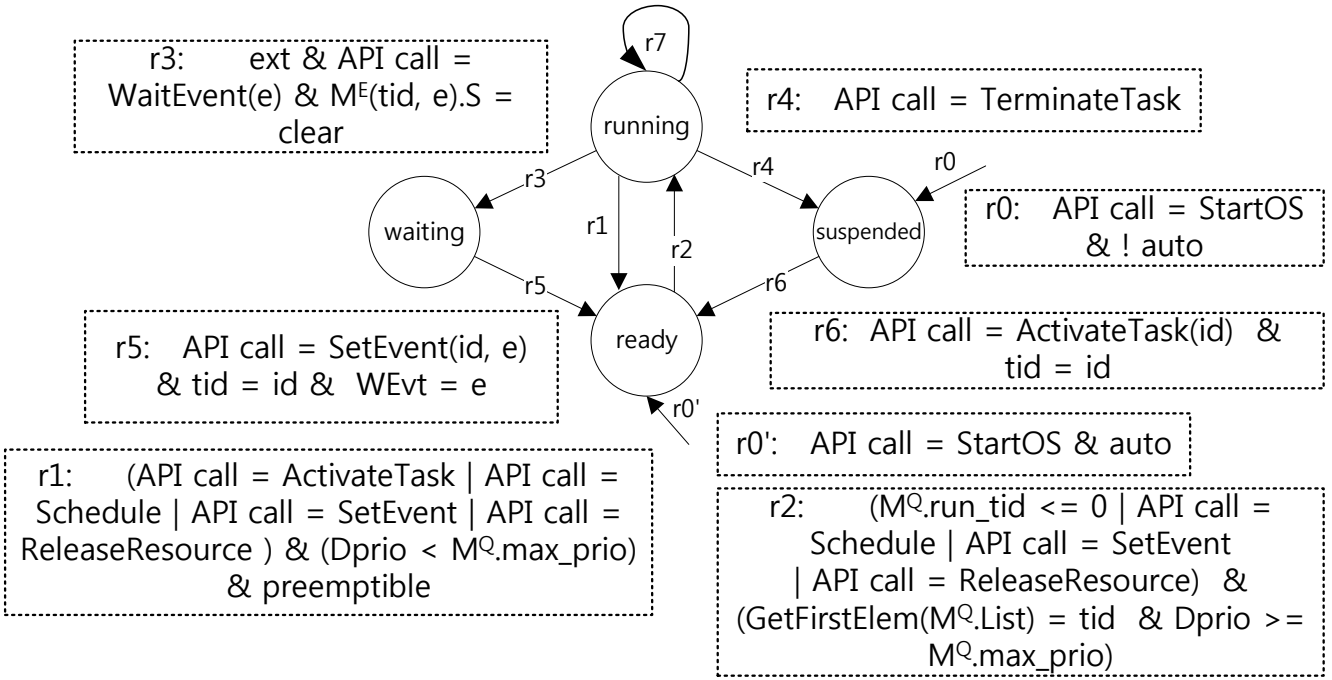


Complexity of code

- 8 major API functions related to Task Management
  - ActivateTask, ChainTask, TerminateTask, Schedule, WaitEvent, SetEvent, GetResource, ReleaseResource



A Task Model: Statemachine representation (1)




```
MODULE Task(env, tid, ptiv, pri, autostrt, extended, rq, e_run, res, evt)
```

A Task Model: Statemachine representation (2)

```
MODULE Task(env, tid, ptiv, pri, autostrt, extended, rq, e_run, res, evt)
VAR
state : {SUS, RDY, WIT, RUN};
prio : 1..5;
go_rdy : boolean;
wait_e : 1..2;
c2 : NotInBetween(env,
c3 : MustEndWith(env,

FROZENVAR
id : 1..5;
autostart : boolean;

DEFINE
exc_set2 := {TT, WE};
end_set3 := {TT};
ext := extended;

ASSIGN
init(autostart) := aut
init(id)        := tid
init(state)     := cas
          autostrt : RDY
          TRUE : SUS;
esac;
init(wait_e)    := 1;
next(wait_e)    := case
          env.nWE & state = RUN & nex    tate) = WIT : next(env.p_e);
          TRUE : wait_e;
esac;
next(state)     := case
          state = RDY : case
          !e_run & prio >= rq.max_prio : RUN;
          (env.nSC | env.nRR | env.nSE) & rq.pq[prio][0] = 1 & prio > rq.max_prio : RUN;
          TRUE : state;
esac;
          state = RUN : case
          next(env.api) = None & ptiv = TRUE & prio < rq.max_prio : RDY;
          env.nAT & ptiv = TRUE & prio < (rq.max_prio) : RDY;
          (env.nSC | env.nSE | env.nRR) & prio < rq.max_prio & ptiv = TRUE : RDY;
          env.nCT : SUS;
          env.nTT : SUS;
          ext & env.nWE & next(evt.valid) & evt.state[tid][next(env.p_e)] = clear : WIT;
          TRUE : state;
esac;
          state = SUS & (env.nAT | env.nCT) & next(env.p_t) = id : {state, RDY};
          state = WIT & env.nSE & next(env.p_t) = tid & evt.valid & wait_e = next(env.p_e) : RDY;
          TRUE : state;
esac;
init(go_rdy)    := (state = RDY);
next(go_rdy)    := case
          state = SUS & next(state) = RDY : TRUE;
          TRUE : FALSE;
esac;
init(prio)      := pri;
next(prio)      := case
          env.nGR & state = RUN & next(res.ceil_prio) > prio : next(res.ceil_prio);
          env.nRR : pri;
          TRUE : prio;
esac;
```

```
next(state) := case
    state = RDY :
      case
        !e_run & prio >= rq.max_prio : RUN;
        (env.nSC | env.nRR | env.nSE) & rq.pq[prio][0] = 1 & prio > rq.max_prio : RUN;
        TRUE : state;
      esac;
    ....
```

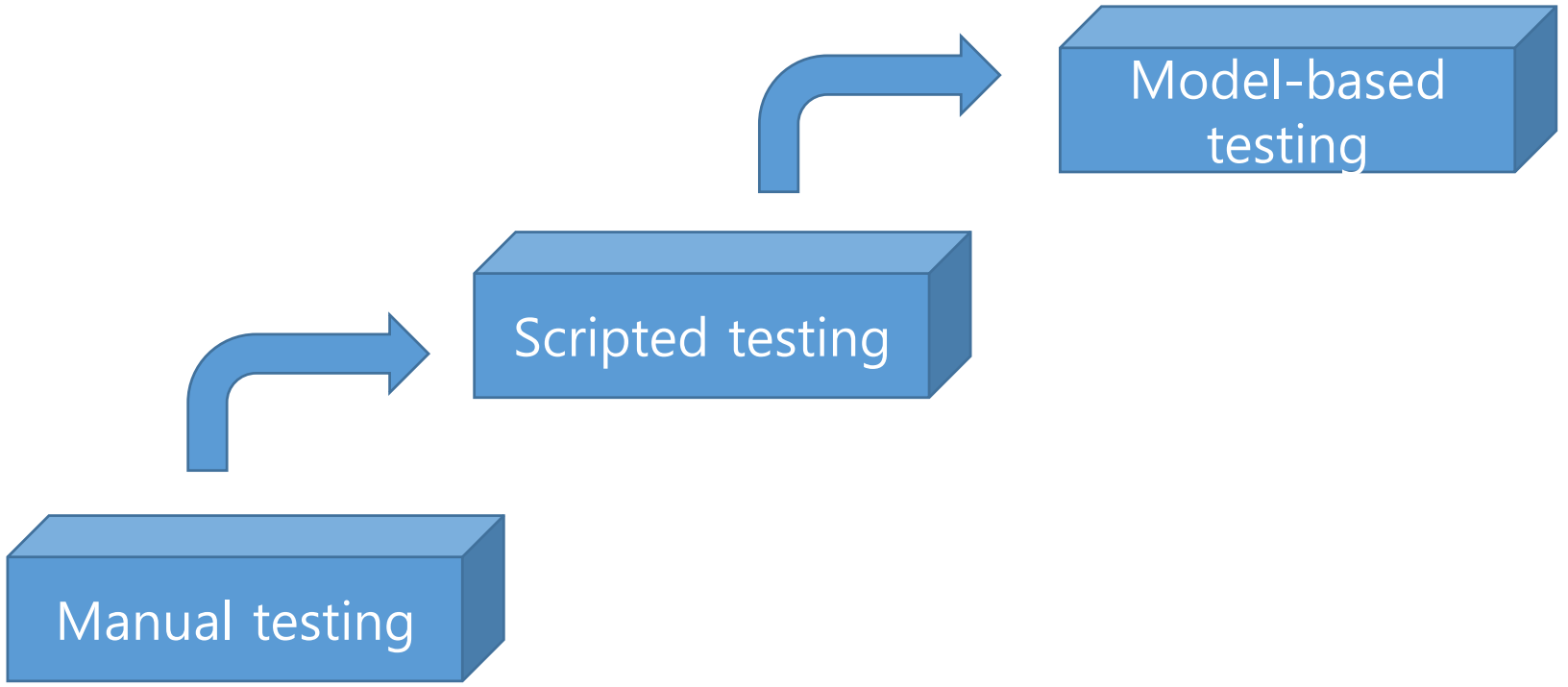
Summary

- Models are useful abstractions
  - In specification and design, they help us think and communicate about complex artifacts by emphasizing key features and suppressing details
  - Models convey structure and help us focus on one thing at a time
- We can use models in systematic testing
- Model-based testing automates testing process at any level o f software development
  - Automation requires the use of formal models

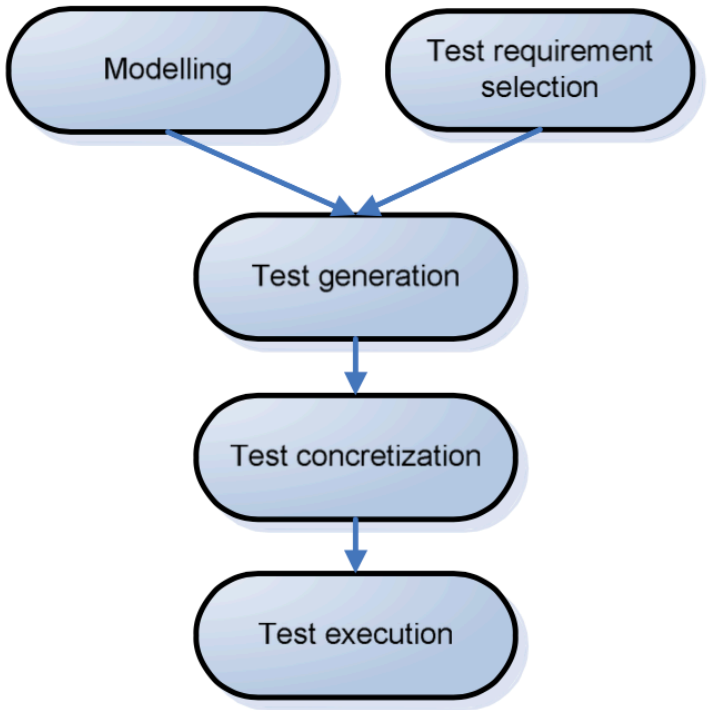
Model-based testing

Concepts

Software testing evolution



Model-based testing

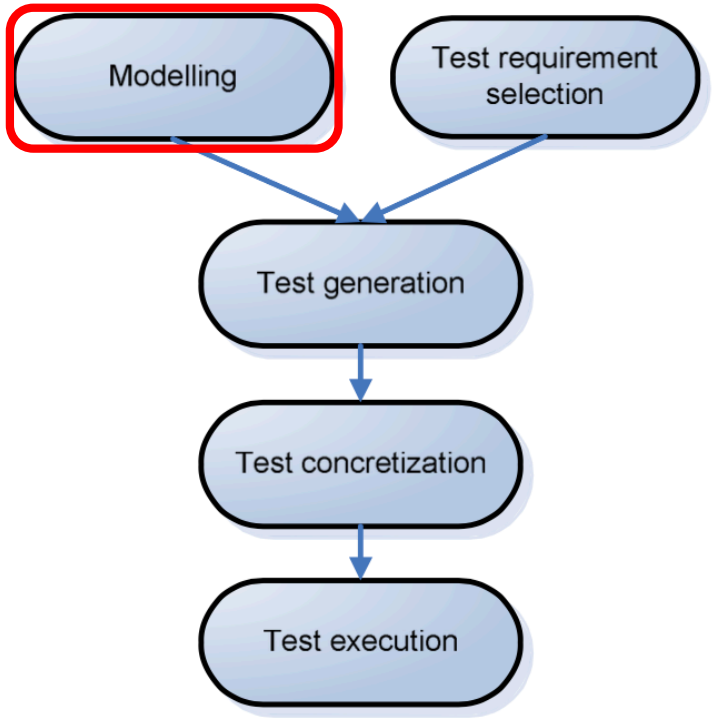


- Model-based testing is software te sting in which test cases are gener ated in whole or in part from a m odel that describes some (function al, non-functional) aspects of the s ystem under test (SUT)
- Almost synonyms
  - Model-driven testing
  - Test generation

Model-based vs. model-driven

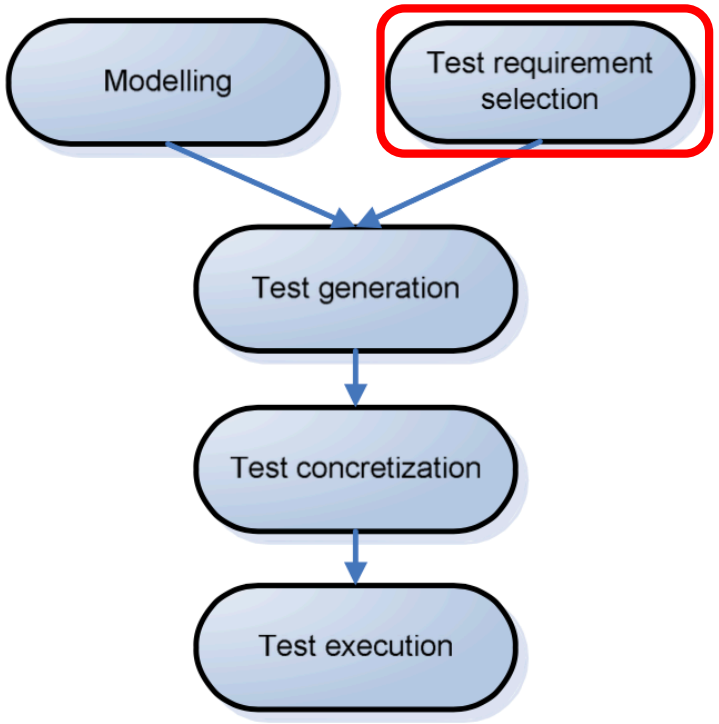
- Model-based
  - Starts from models
- Model-driven
  - Anything can be a model (through abstraction & extraction)

Model-based testing: modeling



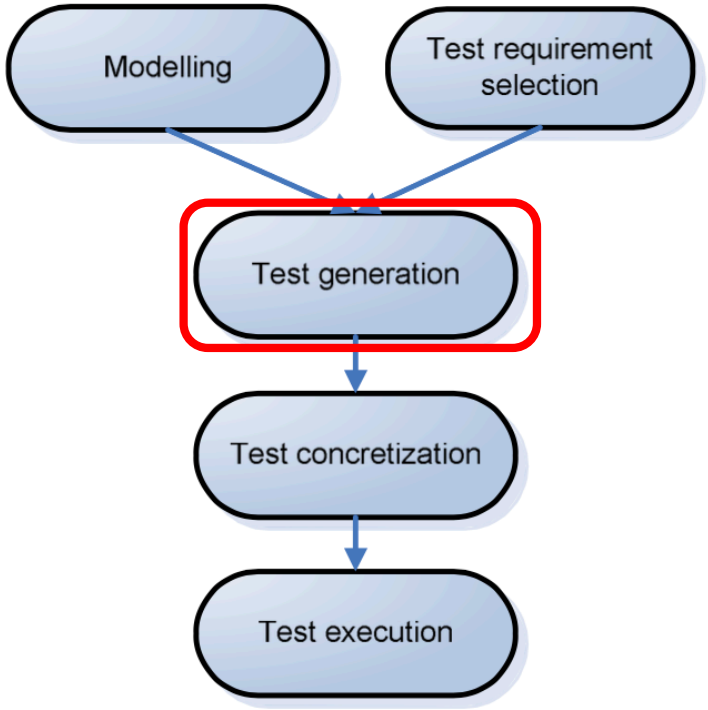
- Purpose: To describe the system requirements for test generator
- Choice of modeling notations
  - General vs. domain-specific
  - Control-oriented vs. data flow oriented

Model-based testing: Test requirements selection



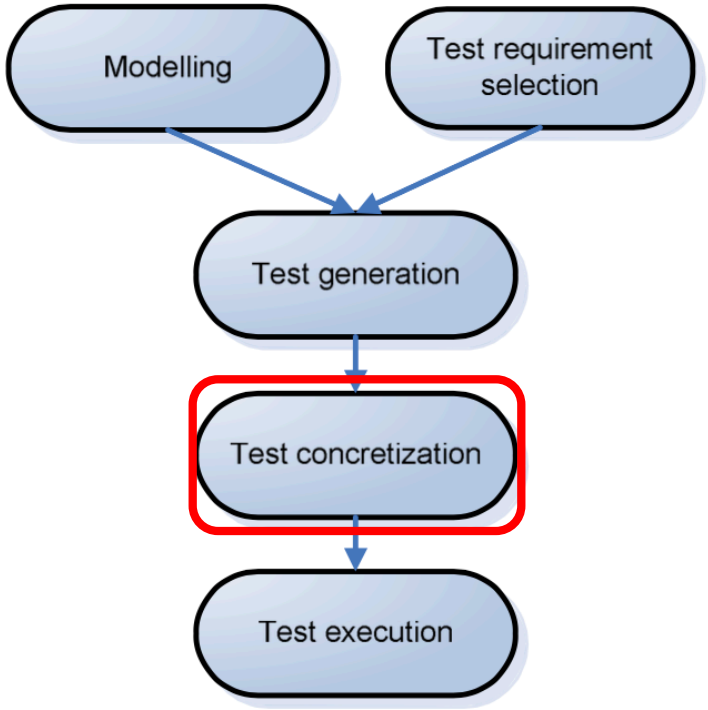
- Purpose: To guide test generation
- Categories:
  - Coverage criteria
    - State coverage
    - Transition coverage
    - Path coverage
  - Walking algorithms
    - Random walking
    - Coverage guided

Model-based testing: Test generation



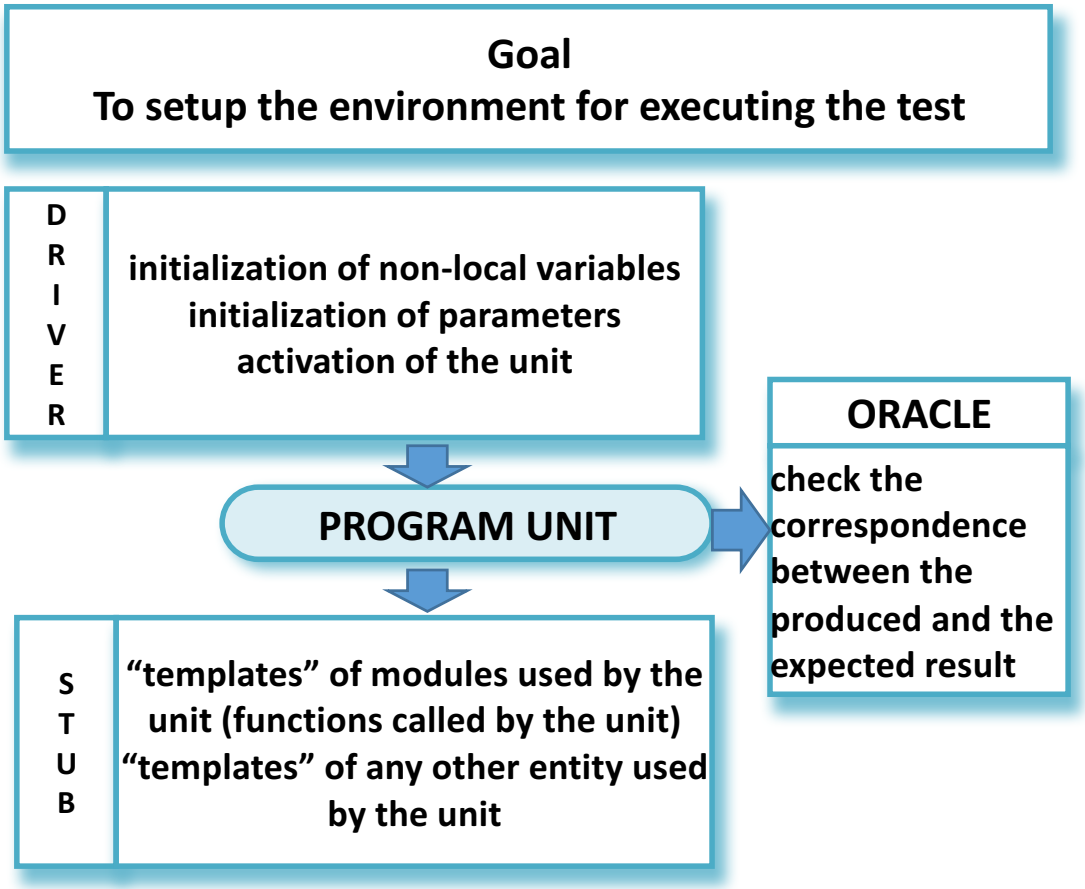
- Purpose: To design the tests
- Use search algorithms
- Tests are written in determined format

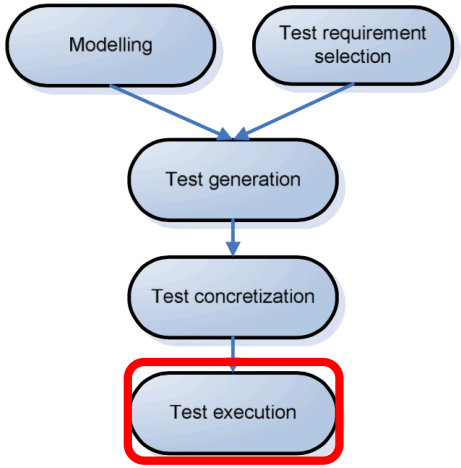
Model-based testing: Test concretizatio n



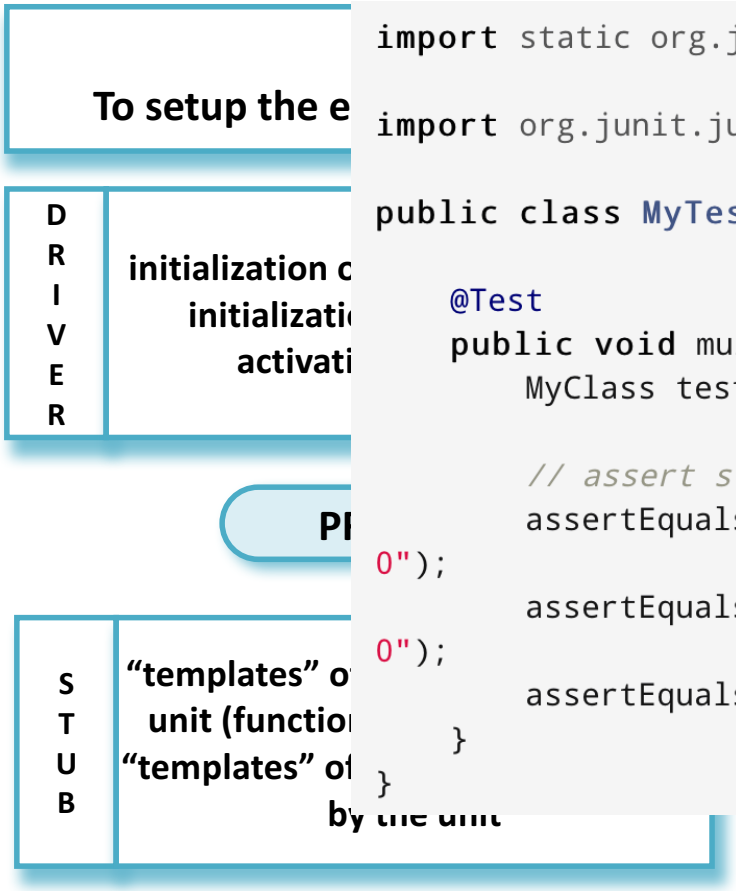
- Purpose: Concretize abstract test suite to executable level
- Tools are provided
  - Various test exporting formats
  - Do-it-yourself plug-ins

Test execution environment





Test execution environment : Junit exam ple



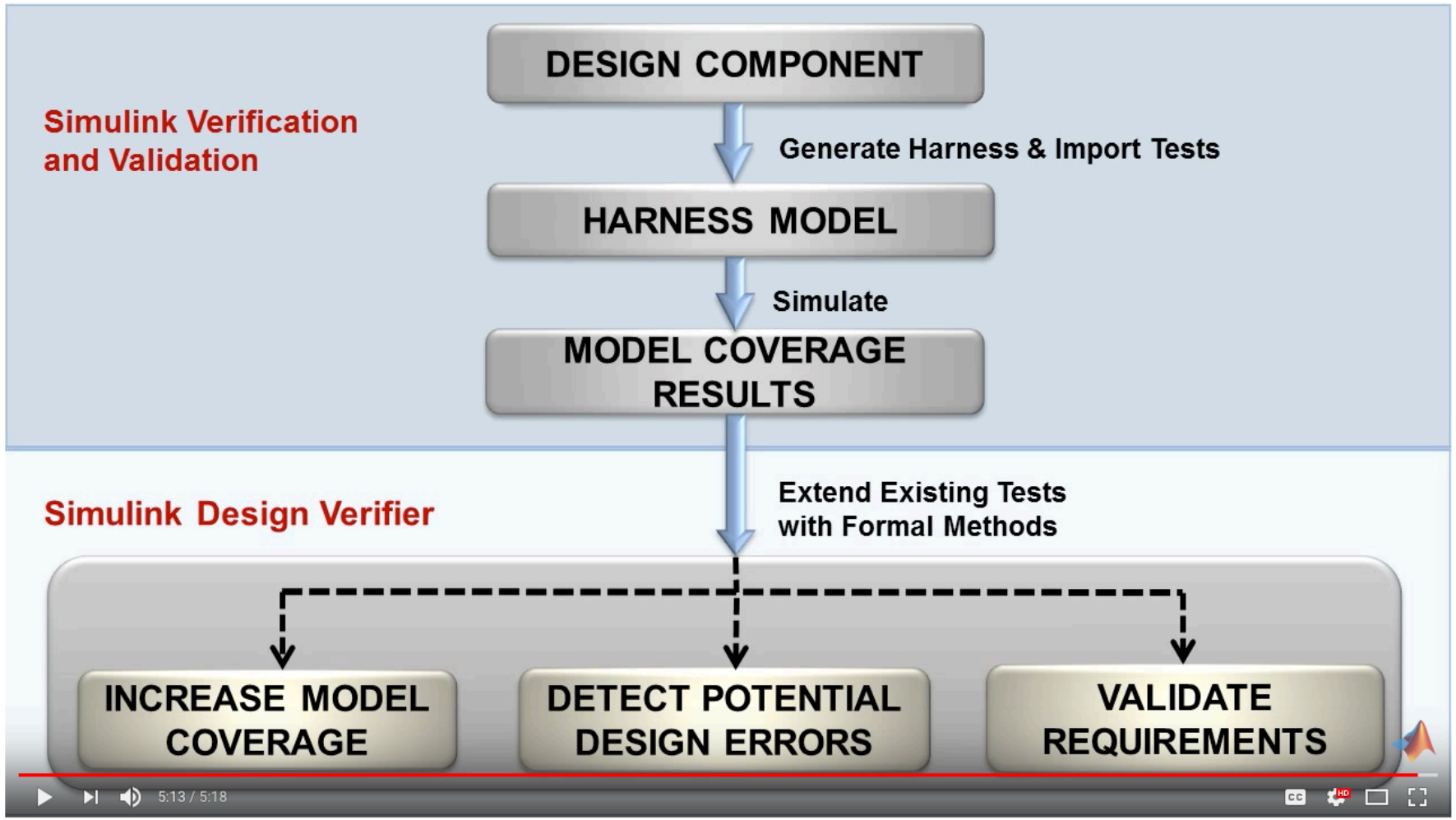
JAVA

```java
import static org.junit.jupiter.api.Assertions.assertEquals;

import org.junit.jupiter.api.Test;

public class MyTests {

    @Test
    public void multiplicationOfZeroIntegersShouldReturnZero() {
        MyClass tester = new MyClass(); // MyClass is tested

        // assert statements
        assertEquals(0, tester.multiply(10, 0), "10 x 0 must be
0");
        assertEquals(0, tester.multiply(0, 10), "0 x 10 must be
0");
        assertEquals(0, tester.multiply(0, 0), "0 x 0 must be 0");
    }
}
```

Model-based testing demo (Mathlab Simulink)

- https://www.youtube.com/watch?v=2bO72KEWLNY

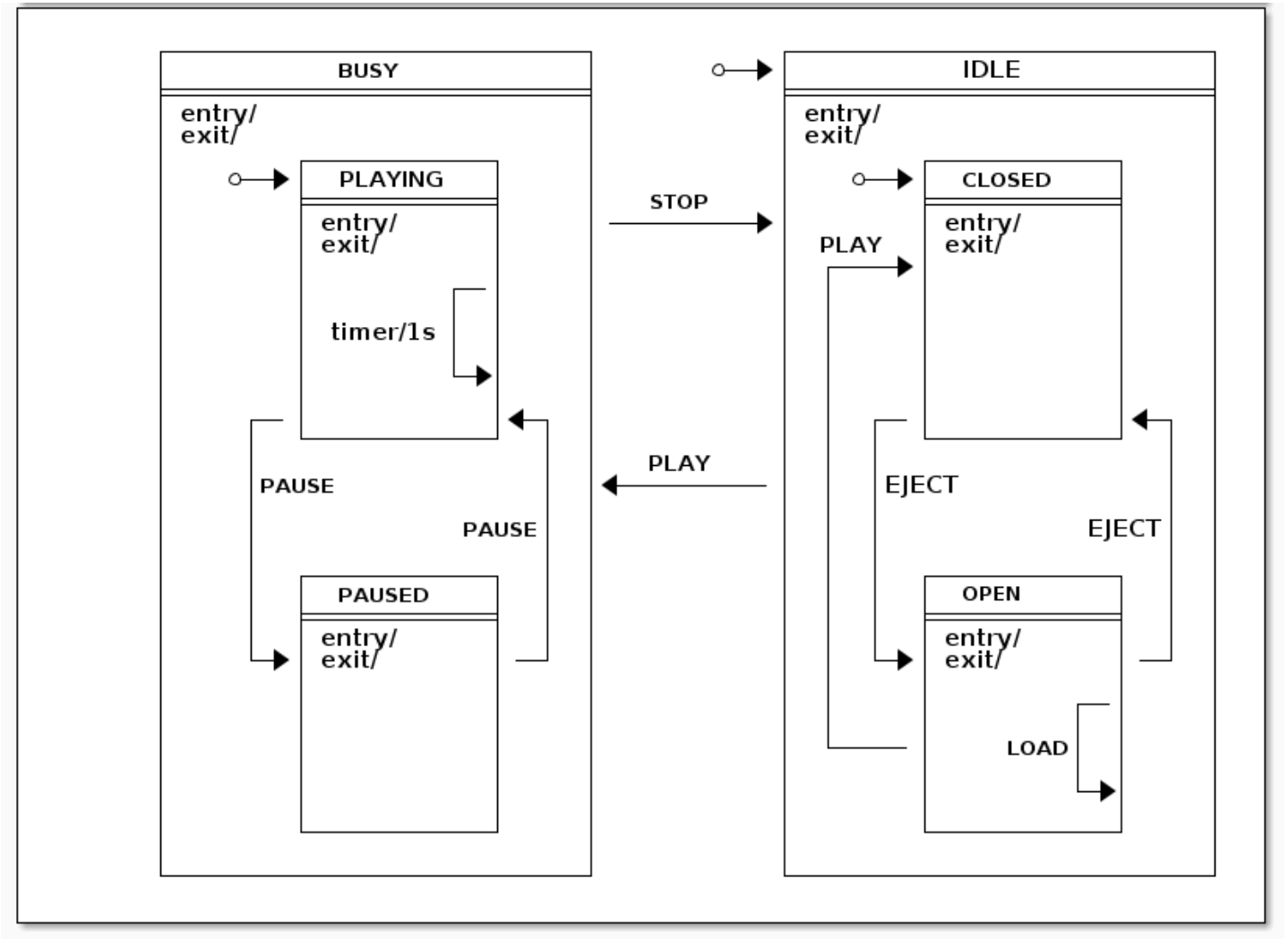
Model-based testing in Simulink



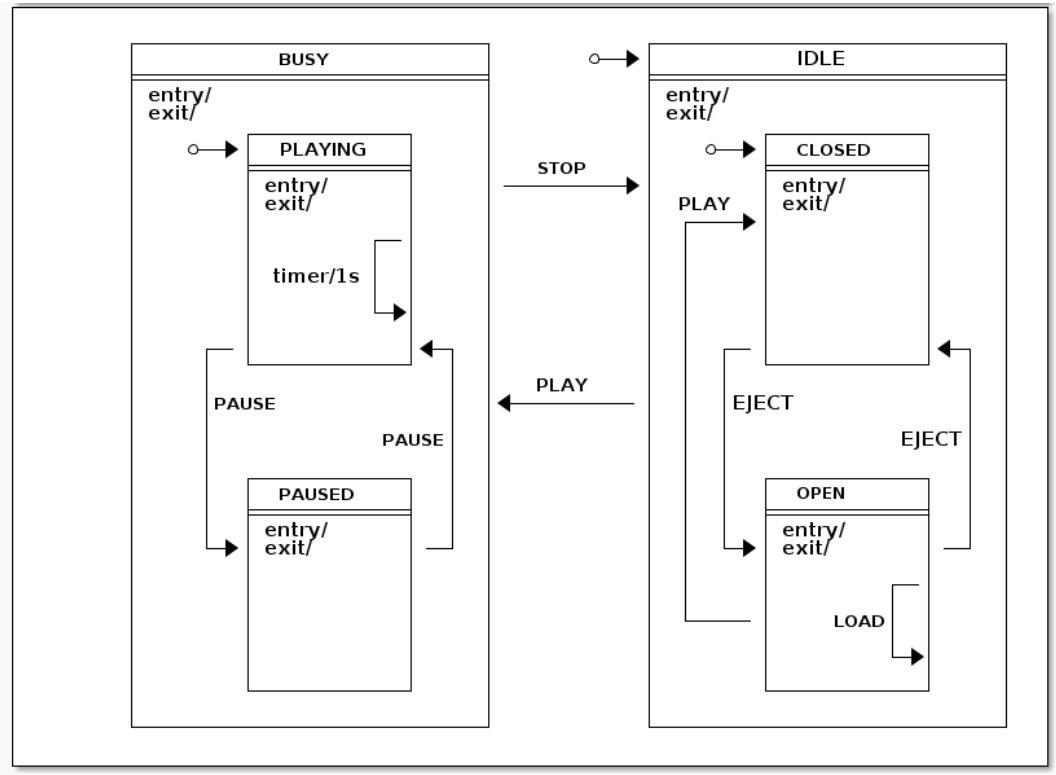
Main benefits of model-based testing

- Easier test suite maintenance
- Automated test design
  - Save effort
- Better test quality
  - No human mistakes
- Tool support for the validation of requirements
- Early detection of design errors
  - If extended by formal methods

Example (1): Test generation from State machine (CD player)



Example (1): Test generation from State machine (CD player)

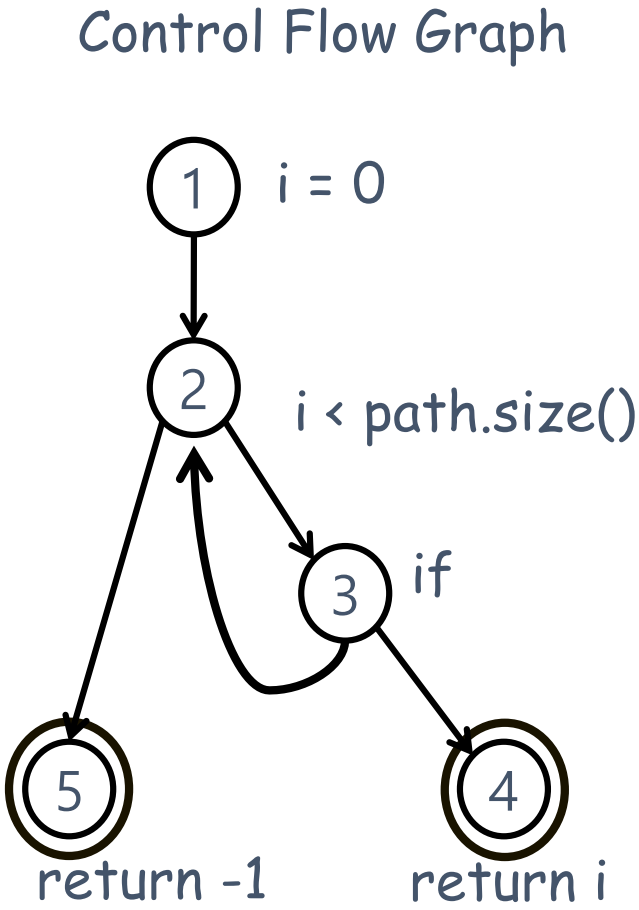


- State coverage
  - Test1: EJECT, OPEN
    PLAY, PLAYING
    PAUSE, PAUSED
- Transition coverage?

Example (2): Test generation from CFG

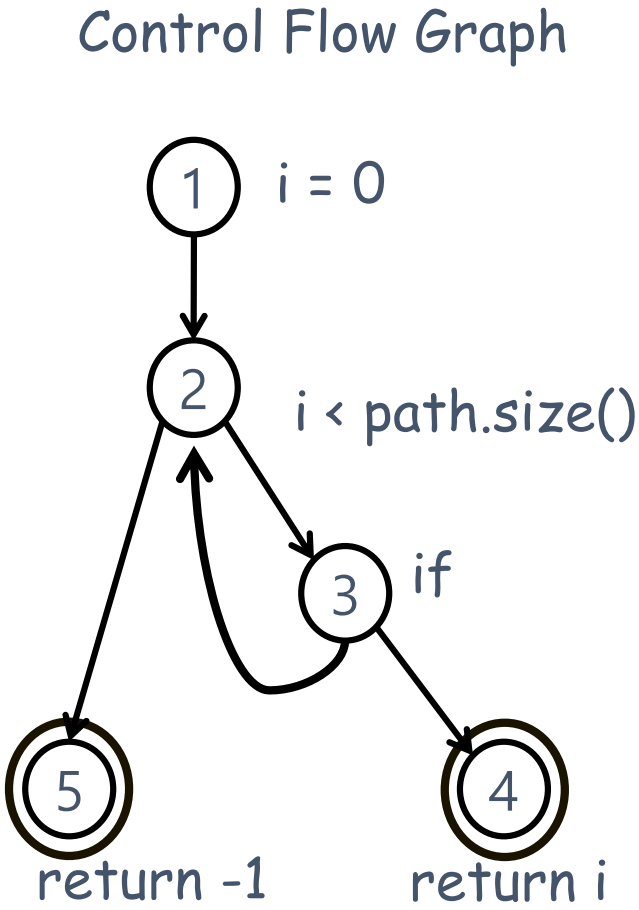
Software Artifact : Java Method

```
/**
 * Return index of node n at the
 * first position it appears,
 * -1 if it is not present
*/
public int indexOf (Node n)
{
  for (int i=0; i < path.size(); i++)
    if (path.get(i).equals(n))
      return i;
  return -1;
}
```



Example (2): Test generation from CFG

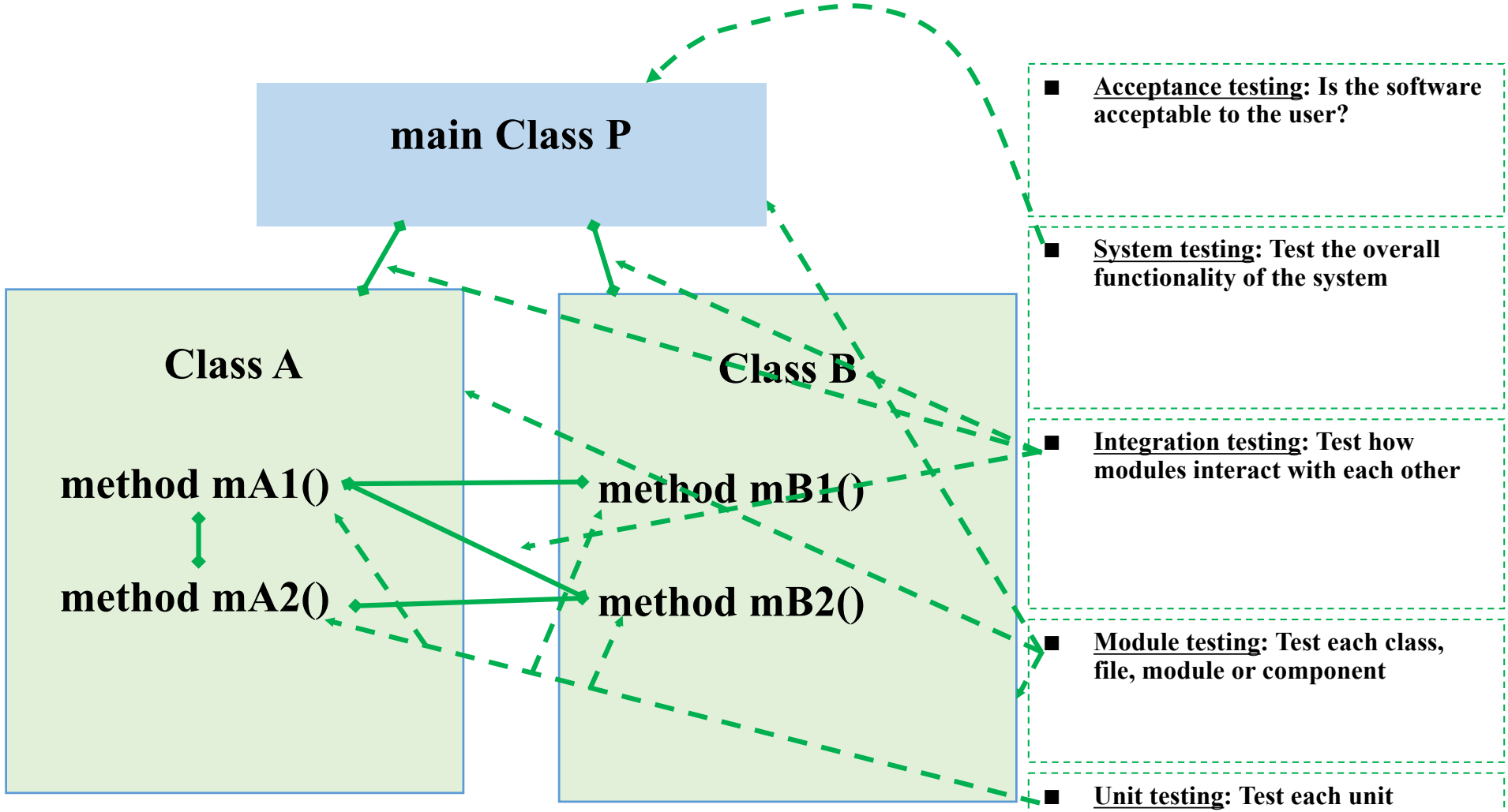
- State coverage
  - Test input (path, n): ({1}, 1)
- Transition coverage?



Changing Notions of Testing

- Old view considered testing at each software development phase to be very different from other phases
  - Unit, module, integration, system …
- New view is in terms of structures and criteria
  - Graphs, logical expressions, syntax, input space
- Test design is largely the same at each phase
  - Creating the model is different
  - Choosing values and automating the tests is different

Old : Testing at Different Levels



This view obscures underlying similarities

New : Test Coverage Criteria

A tester's job is simple :

Define a model of the software, then find ways to cover it

- Test Requirements : Specific things that must be satisfied or covered during testing

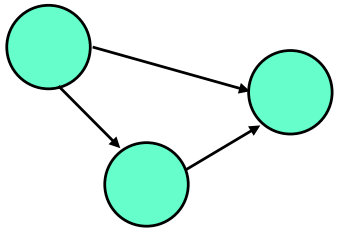
- Test Criterion : A collection of rules and a process that define test requirements

Testing researchers have defined dozens of criteria, but they are all really just a few criteria on four types of structures ...

Criteria Based on Structures

Structures : Four ways to model software

1. Graphs



2. Logical Expressions

```
(not X or not Y) and A and B
```

3. Input Domain Characterization

```
A: {0, 1, >1}
B: {600, 700, 800}
C: {swe, cs, isa, infs}
```

4. Syntactic Structures

```
if (x > y)
    z = x - y;
else
   z = 2 * x;
```

Source of Structures

- These structures can be extracted from lots of software artifacts
  - Graphs can be extracted from UML use cases, finite state machines, source code, …
  - Logical expressions can be extracted from decisions in program source, guards on transitions, conditionals in use cases, …

- "*model-based testing*" and "model-driven testing" are different (Jeff Offutt)
  - Model-based testing derives tests from a model that describes some aspects of the system under test
  - The source is usually *not* considered a model

Summary

- Test design is largely the same at each phase
- Tester's job is to define a model of the software, then find ways to cover it
- There are four ways to model software
  - Using graphs, logical expression, input domain characterization, and syntactic structures

Model-based testing techniques

Deriving test cases from finite state machines

From informal specifications..

Maintenance: The Maintenance function records the history of items undergoing maintenance.

If the product is covered by warranty or maintenance contract, maintenance can be requested either by calling the maintenance toll free number, or through the web site, or by bringing the item to a designated maintenance station.

If the maintenance is requested by phone or web site and the customer is a US or EU resident, the item is picked up at the customer site, otherwise, the customer shall ship the item with an express courier.

If the maintenance contract number provided by the customer is not valid, the item follows the procedure for items not covered by warranty.

If the product is not covered by warranty or maintenance contract, maintenance can be requested only by bringing the item to a maintenance station. The maintenance station informs the customer of the estimated costs for repair. Maintenance starts only when the customer accepts the estimate.

If the customer does not accept the estimate, the product is returned to the customer.

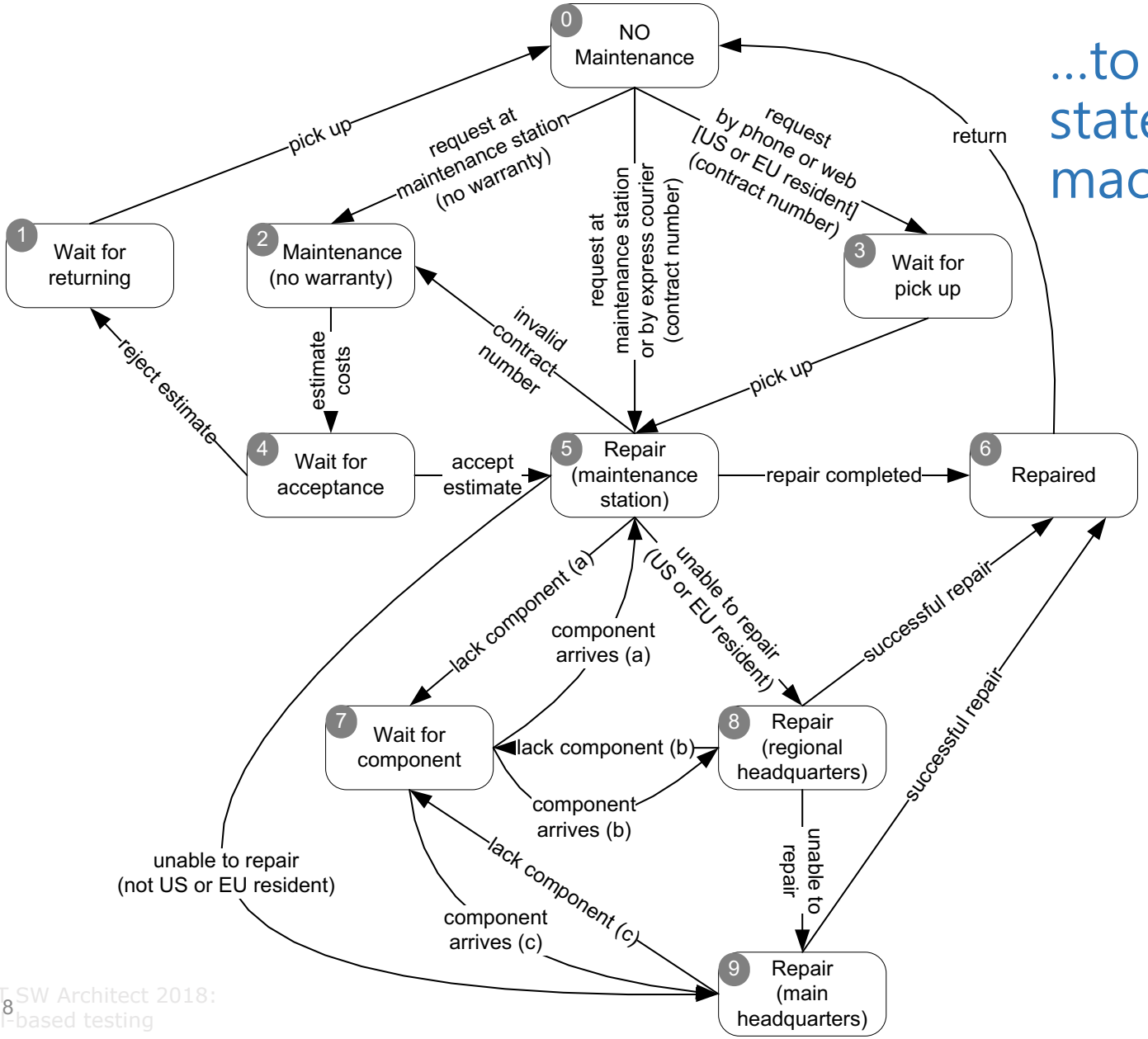
Small problems can be repaired directly at the maintenance station. If the maintenance station cannot solve the problem, the product is sent to the maintenance regional headquarters (if in US or EU) or to the maintenance main headquarters (otherwise).

If the maintenance regional headquarters cannot solve the problem, the product is sent to the maintenance main headquarters.

Maintenance is suspended if some components are not available.

Once repaired, the product is returned to the customer.

Deriving test cases from finite state machines



...to a finite state machine...

Deriving test cases from finite state machines

...to a test suite

TC1	0	2	4	1	0						
TC2	0	5	2	4	5	6	0				
TC3	0	3	5	9	6	0					
TC4	0	3	5	7	5	8	7	8	9	6	0

Meaning: From state 0 to state 2 to state 4 to state 1 to state 0

Is this a thorough test suite?
How can we judge?

Deriving test cases from finite state machines

"Covering" finite state machines

- State coverage:
  - Every state in the model should be visited by at least one test case
- Transition coverage
  - Every transition between states should be traversed by at least one test case.
  - *This is the most commonly used criterion*
    - A transition can be thought of as a (precondition, postcondition) pair

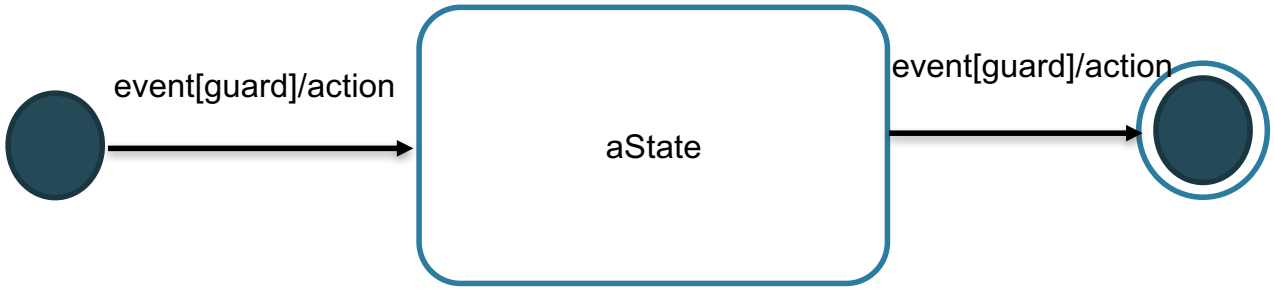
Deriving test cases from finite state machines

Path sensitive criteria?

- Basic assumption: States fully summarize history
  - No distinction based on how we reached a state; this should be true of well-designed state machine models
- If the assumption is violated, we may distinguish paths and devise criteria to cover them
  - Single state path coverage:
    - traverse each subpath that reaches each state at most once
  - Single transition path coverage:
    - Traverse each transition at most once
  - Boundary interior loop coverage:
    - each distinct loop of the state machine must be exercised the minimum, an intermediate, and the maximum or a large number of times

Deriving test cases from finite state machines

Finite statemachine notation

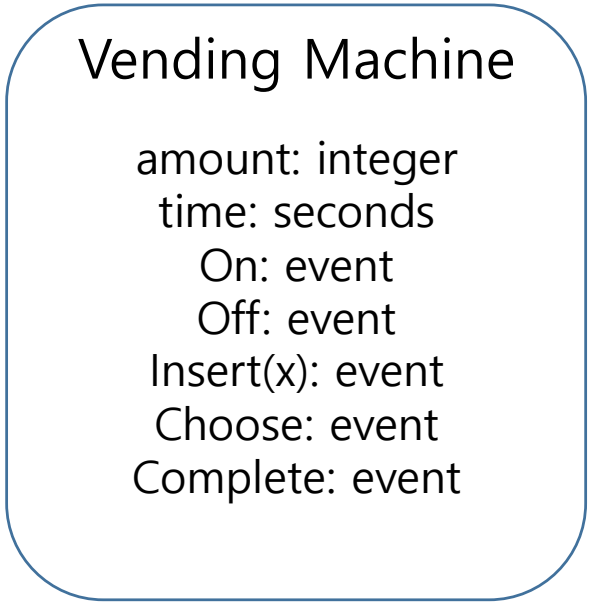


Exercise: Chocolate Vending machine

- The vending machine sells chocolates
- The price of a chocolate is 500 Won
- When the machine is turned on, it waits for a customer to ins ert coins or paper money
- If 500 Won or more is inserted, the button for choosing choc olate is illuminated
- If there is no more actions after inserting money for more th an 10 seconds, it returns the money already inserted
- If the customer presses the illuminated button, it moves the c hocolate to the exit and returns the remaining money

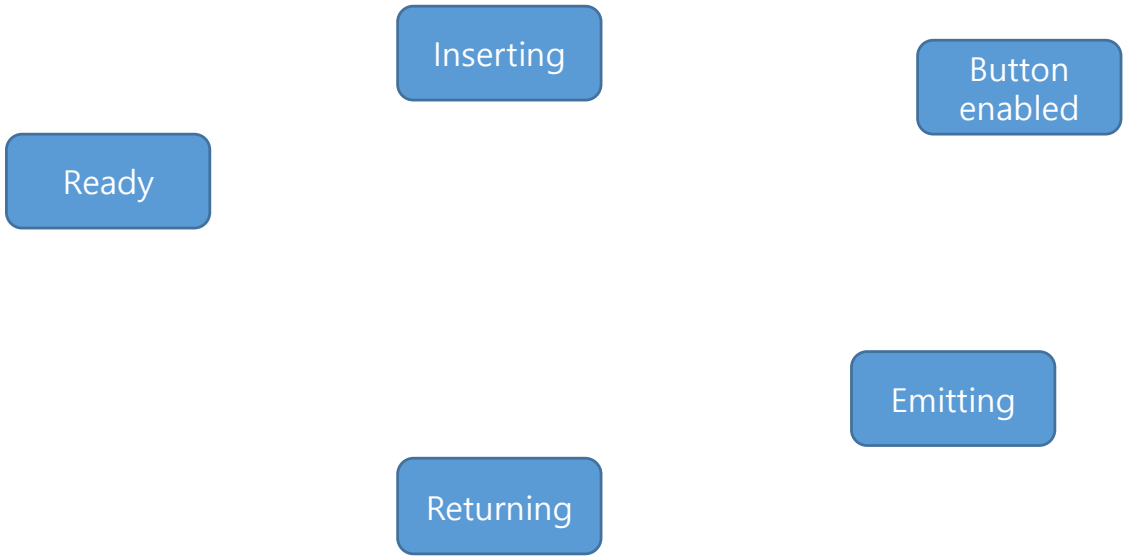
Exercise: Chocolate Vending machine

- Identifying data and operations



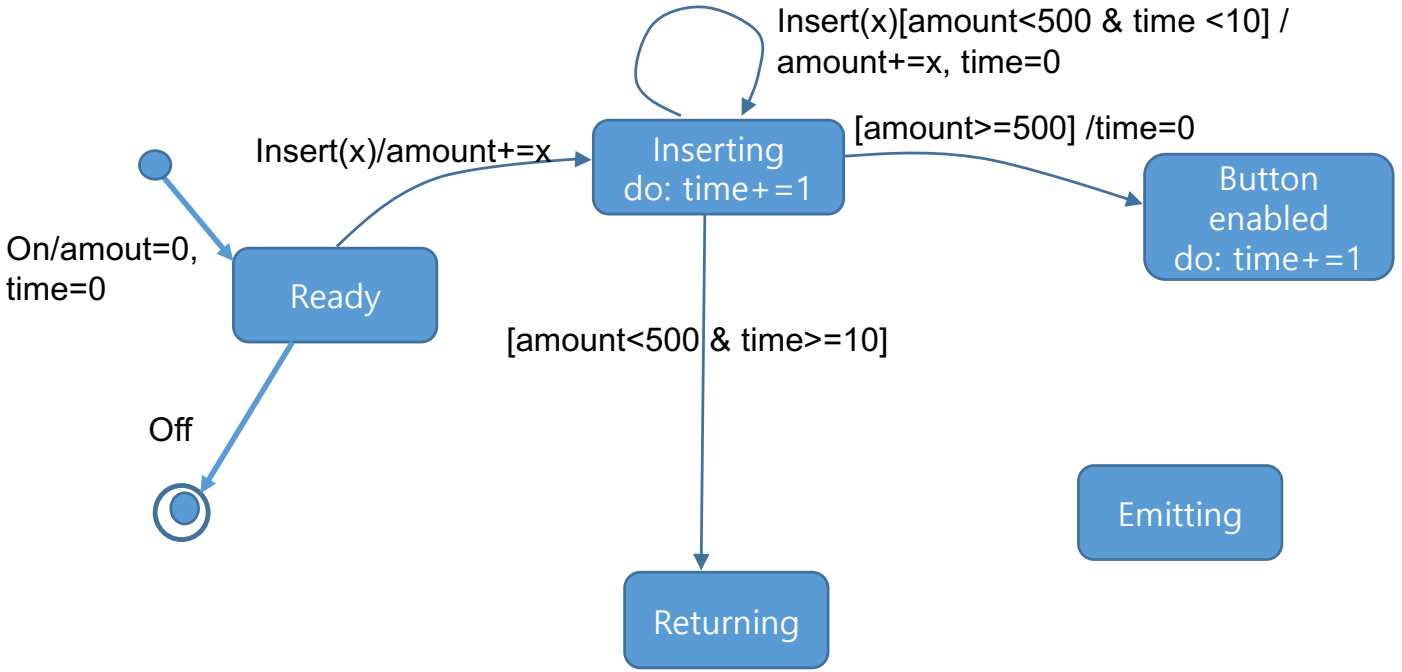
Exercise: Chocolate Vending machine

- Identifying states



Exercise: Chocolate Vending machine

- Deciding transitions



Exercise: Chocolate Vending machine

- Deriving test cases
- State coverage
  - Test sequence 1
    - Turn machine on → ready
    - Insert(1000) → Inserting, amount =500 → button enabled
    - Choose within 5 seconds → Emitting, amount = 500 → returning → ready
- Transition coverage
  - Test sequence 2

Testing decision structures

- Some specifications are structured as decision tables, decision trees, or flow charts.
- We can exercise these as if they were program source code.

Testing decision structures

from an informal specification..

Pricing: The pricing function determines the adjusted price of a configuration for a particular customer.

The scheduled price of a configuration is the sum of the scheduled price of the model and the scheduled price of each component in the configuration. The adjusted price is either the scheduled price, if no discounts are applicable, or the scheduled price less any applicable discounts.

There are three price schedules and three corresponding discount schedules, Business, Educational, and Individual.

….

- Educational prices: The adjusted price for a purchase charged to an educational account in good standing is the scheduled price from the educational price schedule. No further discounts apply.

…

- Special-price non-discountable offers: Sometimes a complete configuration is offered at a special, non-discountable price. When a special, non-discountable price is available for a configuration, the adjusted price is the non-discountable price or the regular price after any applicable discounts, whichever is less

Testing decision structures

…to a decision table …

	edu		**individual**					
EduAc	T	T	F	F	F	F	F	F
BusAc	-	-	F	F	F	F	F	F
CP > CT1	-	-	F	F	T	T	-	-
YP > YT1	-	-	-	-	-	-	-	-
CP > CT2	-	-	-	-	F	F	T	T
YP > YT2	-	-	-	-	-	-	-	-
SP < Sc	F	T	F	T	-	-	-	-
SP < T1	-	-	-	-	F	T	-	-
SP < T2	-	-	-	-	-	-	F	T
out	Edu	SP	ND	SP	T1	SP	T2	SP

Testing decision structures

…with constraints…

at-most-one (EduAc, BusAc)

at-most-one (YP < YT1, YP > YT2)

YP > YT2 -> YP > YT1

at-most-one (CP < CT1, CP > CT2)

CP > CT2 -> CP > CT1

at-most-one (SP < T1, SP > T2)

SP > T2 -> SP > T1

Testing decision structures

…to test cases

- Basic condition coverage
  - a test case specification for each column in the table
- Compound condition adequacy criterion
  - a test case specification for each combination of truth values of basic conditions
- Modified condition/decision adequacy criterion (MC/DC)
  - each column in the table represents a test case specification.
  - we add columns that differ in one input row and in outcome, then merge compatible columns
    - Each condition should effect the decision outcome independently
  - Test important combinations of conditions and limit testing costs

Testing decision structures : MC/DC

	C.1	C.1a	C.1b	C.10
EduAc	T	F	T	-
BusAc	-	-	-	T
CP > CT1	-	-	-	F
YP > YT1	-	-	-	F
CP > CT2	-	-	-	-
YP > YT2	-	-	-	-
SP > Sc	F	F	T	T
SP > T1	-	-	-	-
SP > T2	-	-	-	-
out	Edu	*	*	SP

Generate C.1a and C.1b by flipping one element of C.1

C.1b can be merged with an existing column (C.10) in the spec

Outcome of generated columns must differ from source column

Exercise: a&b&c

Test case	a	b	c	out
1	T	T	T	T
2	T	T	F	F
3	T	F	T	F
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

Flowgraph based testing

- If the specification or model has both decisions and sequential logic, we can cover it like program source code.

Flowgraph based testing

from an informal spec (i/iii)...

- Process shipping order: The Process shipping order function checks the validity of orders and prepares the receipt
  A valid order contains the following data:
  - cost of goods: If the cost of goods is less than the minimum processable order (MinOrder) then the order is invalid.
  - shipping address: The address includes name, address, city, postal code, and country.
  - preferred shipping method: If the address is domestic, the shipping method must be either land freight, expedited land freight, or overnight air; If the address is international, the shipping method must be either air freight, or expedited air freight.

Flowgraph based testing

…(ii/iii)…

- a shipping cost is computed based on
  - address and shipping method.
  - type of customer which can be individual, business, educational
- preferred method of payment. Individual customers can use only credit cards, business and educational customers can choose between credit card and invoice
- card information: if the method of payment is credit card, fields credit card number, name on card, expiration date, and billing address, if different than shipping address, must be provided. If credit card information is not valid the user can either provide new data or abort the order

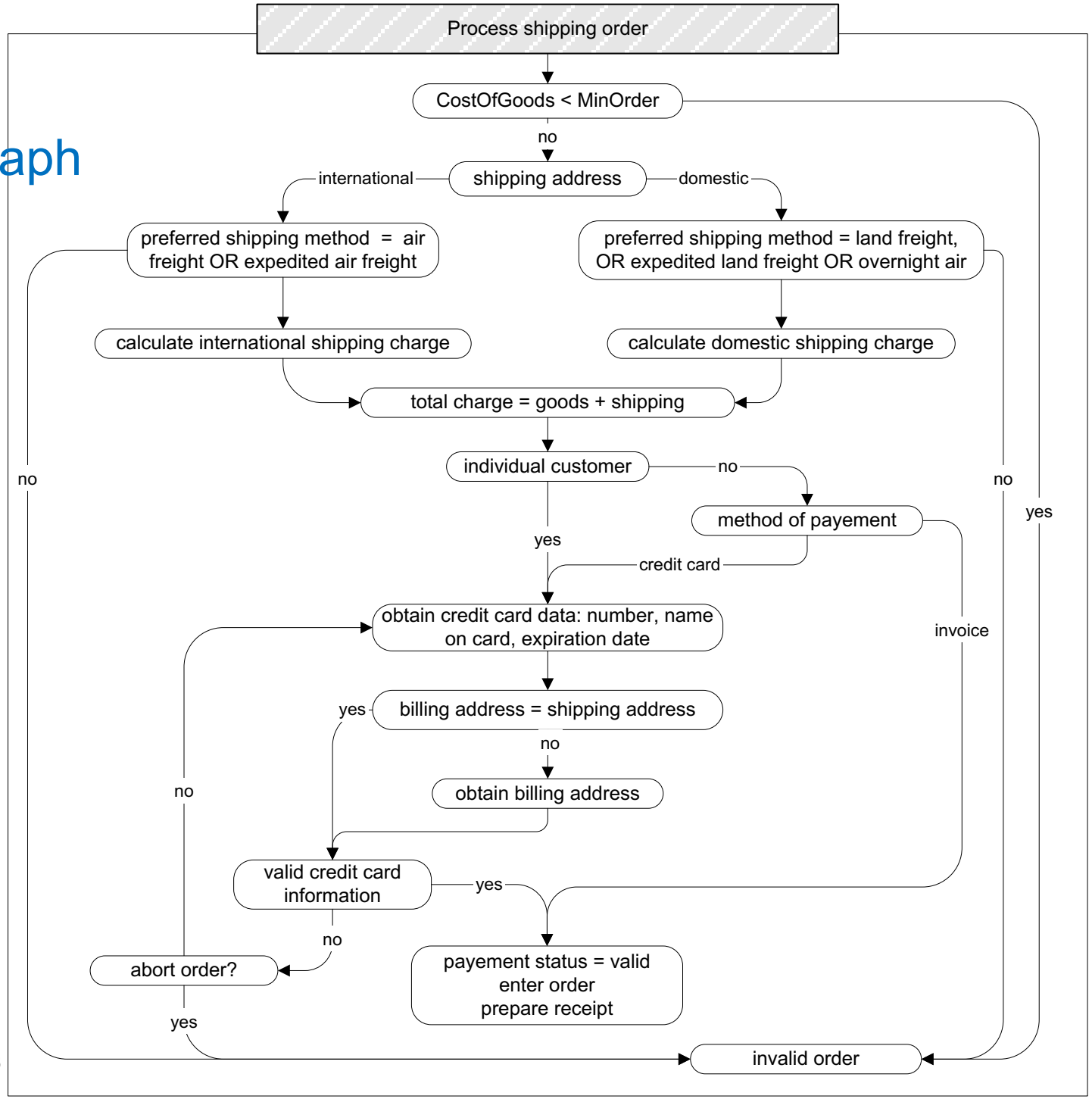
Flowgraph based testing

…(iii/iii)

- The outputs of Process shipping order are
- validity: Validity is a Boolean output which indicates whether the order can be processed.
- total charge: The total charge is the sum of the value of goods and the computed shipping costs (only if validity = true).
- payment status: if all data are processed correctly and the credit card information is valid or the payment is invoice, payment status is set to valid, the order is entered and a receipt is prepared; otherwise validity = false.

Flow graph based testing

…to a flowgraph

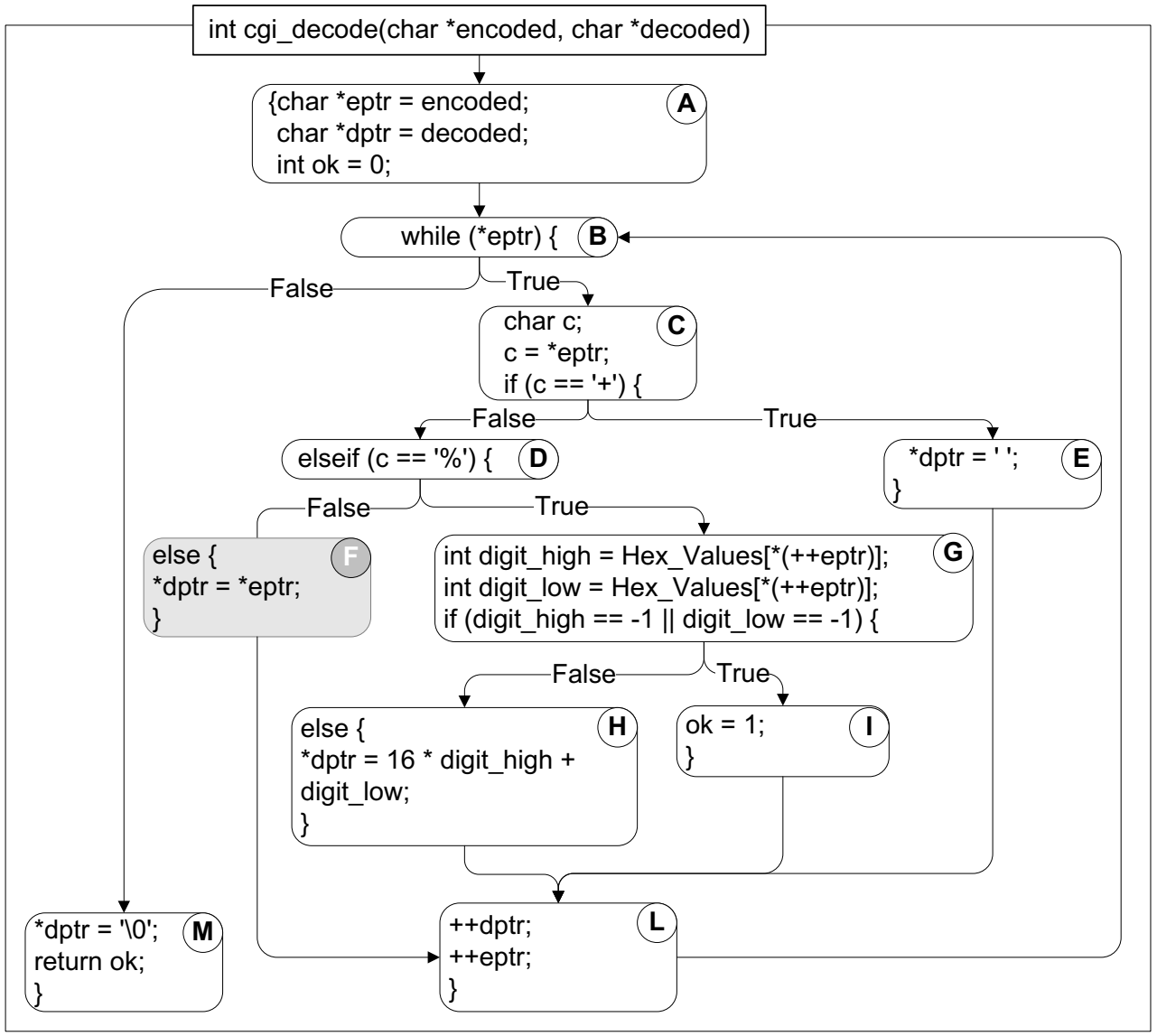


…from the flow graph to test cases

Branch testing: cover all branches

Case	Too Small	Ship Where	Ship Method	Cust Type	Pay Method	Same Address	CC valid
TC-1	No	Int	Air	Bus	CC	No	Yes
TC-2	No	Dom	Land	-	-	-	-
TC-3	Yes	-	-	-	-	-	-
TC-4	No	Dom	Air	-	-	-	-
TC-5	No	Int	Land	-	-	-	-
TC-6	No	-	-	Edu	Inv	-	-
TC-7	No	-	-	-	CC	Yes	-
TC-8	No	-	-	-	CC	-	No (abort)
TC-9	No	-	-	-	CC	-	No (no abort)

Exercise


int cgi_decode(char *encoded, char *decoded)
{char *eptr = encoded;
char *dptr = decoded;
int ok = 0;
A
while (*eptr) {
B
False
True
char c;
c = *eptr;
if (c == '+') {
C
False
True
elseif (c == '%') {
D
*dptr = ' ';
}
E
False
True
else {
*dptr = *eptr;
}
F
int digit_high = Hex_Values[*(++eptr)];
int digit_low = Hex_Values[*(++eptr)];
if (digit_high == -1 || digit_low == -1) {
G
False
True
else {
*dptr = 16 * digit_high +
digit_low;
}
H
ok = 1;
}
I
*dptr = '\0';
return ok;
}
M
++dptr;
++eptr;
}
L

Grammar-based testing

- Complex input is (or can) often be described by a context-free grammar

Grammars in specifications

- Grammars are good at:
  - Representing inputs of varying and unbounded size
  - With recursive structure
  - And boundary conditions
- Examples:
  - Complex textual inputs
  - Trees (search trees, parse trees, …)
    - Note XML and HTMl are trees in textual form
  - Program structures
    - Which are also tree structures in textual format!

Grammar-based testing

- Test cases are strings *generated* from the grammar
- Coverage criteria:
  - Production coverage: each production must be used to generate at least one (section of) test case
  - Boundary condition: annotate each recursive production with minimum and maximum number of application, then generate:
    - Minimum
    - Minimum + 1
    - Maximum - 1
    - Maximum

Grammar-based testing

from an informal specification (i/iii)…

- The Check-configuration function checks the validity of a computer configuration.
- The parameters of check-configuration are:
  - Model
  - Set of components

Grammar-based testing

… (ii/iii)…

- Model: A model identifies a specific product and determines a set of constraints on available components. Models are characterized by logical slots for components, which may or may not be implemented by physical slots on a bus. Slots may be required or optional. Required slots must be assigned with a suitable component to obtain a legal configuration, while optional slots may be left empty or filled depending on the customers' needs
  - Example: The required ``slots'' of the Chipmunk C20 laptop computer include a screen, a processor, a hard disk, memory, and an operating system. (Of these, only the hard disk and memory are implemented using actual hardware slots on a bus.) The optional slots include external storage devices such as a CD/DVD writer.

Grammar-based testing

… (iii/iii)

- Set of Components: A set of [slot,component] pairs, which must correspond to the required and optional slots associated with the model. A component is a choice that can be varied within a model, and which is not designed to be replaced by the end user. Available components and a default for each slot is determined by the model. The special value empty is allowed (and may be the default selection) for optional slots. In addition to being compatible or incompatible with a particular model and slot, individual components may be compatible or incompatible with each other.
  - Example: The default configuration of the Chipmunk C20 includes 20 gigabytes of hard disk; 30 and 40 gigabyte disks are also available. (Since the hard disk is a required slot, empty is not an allowed choice.) The default operating system is RodentOS 3.2, personal edition, but RodentOS 3.2 mobile server edition may also be selected. The mobile server edition requires at least 30 gigabytes of hard disk.

Grammar-based testing

...to a grammar

<Model>	::= <modelNumber> <compSequence> <optCompSequence>
<compSequence>	::= <Component> <compSequence> | empty
<optCompSequence>	::= <OptionalComponent> <optCompSequence> | empty
<Component>	::= <ComponentType> <ComponentValue>
<OptionalComponent>	::= <ComponentType>
<modelNumber>	::= string
<ComponentType>	::= string
<ComponentValue>	::= string

Grammar-based testing

...to a grammar with limits

Model	<Model>	::= <modelNumber> <compSequence> <optCompSequence>
compSeq1 [0, 16]	<compSequence>	::= <Component> <compSequence>
compSeq2	<compSequence>	::= empty
optCompSeq1 [0, 16]	<optCompSequence>	::= <OptionalComponent> <optCompSequence>
optCompSeq2	<optCompSequence>	::= empty
Comp	<Component>	::= <ComponentType> <ComponentValue>
OptComp	<OptionalComponent>	::= <ComponentType>
modNum	<modelNumber>	::= string
CompTyp	<ComponentType>	::= string
CompVal	<ComponentValue>	::= string

Grammar-based testing

…to test cases

- "Mod000"
  - Covers Model, compSeq1[0], compSeq2, optCompSeq1[0], optCompSeq2, modNum
- "Mod000 (Comp000, Val000) (OptComp000)"
  - Covers Model, compSeq1[1], compSeq2, optCompSeq2[0], optCompSeq2, Comp, OptComp, modNum, CompTyp, CompVal
- etc…
- Comments:
  - By first applying productions with nonterminals on the right side, we obtain few, large test cases
  - By first applying productions with terminals on the right side, we obtain many, small test cases

Summary

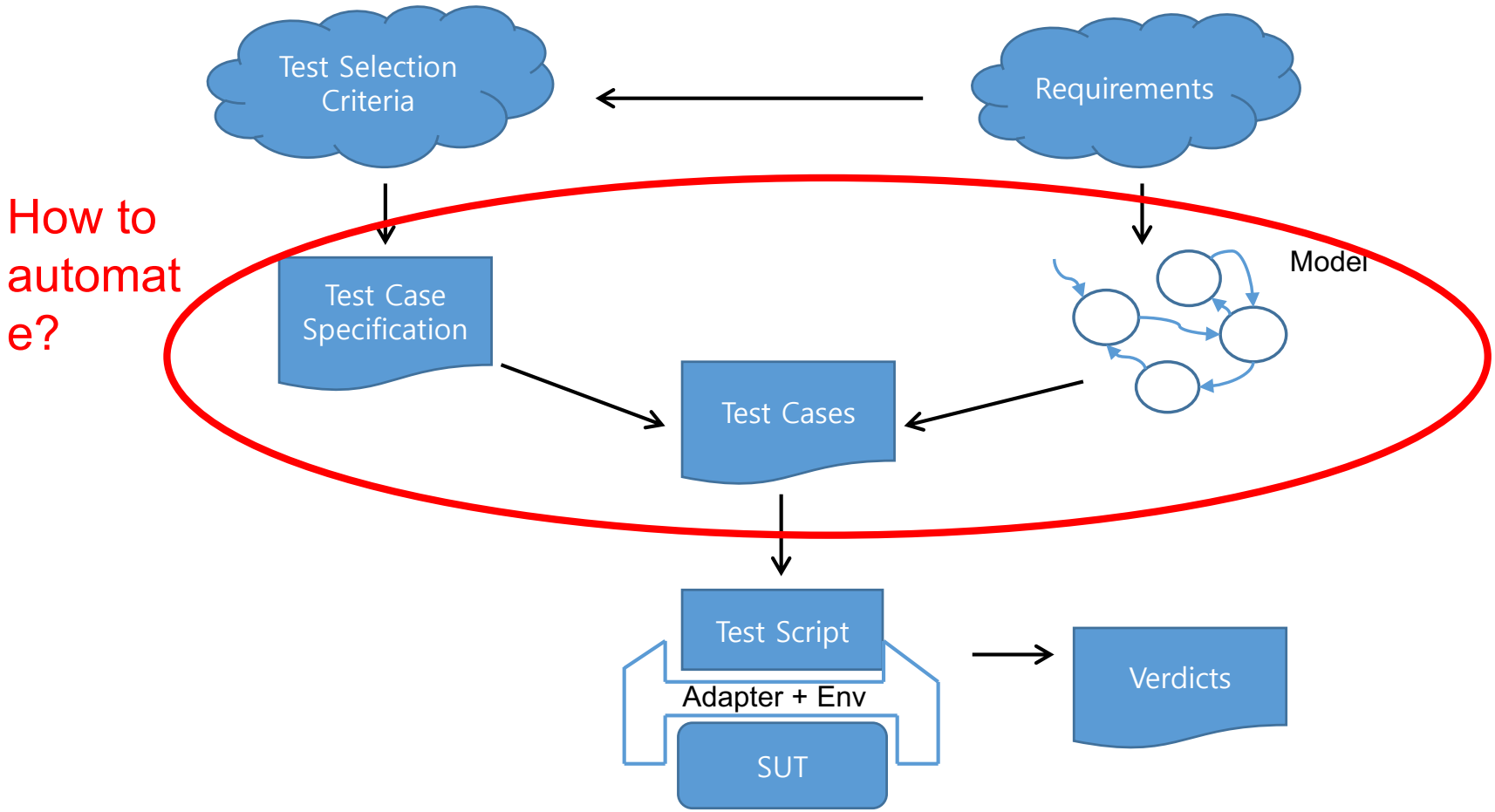
- Models are useful abstractions
  - In specification and design, they help us think and communicate about complex artifacts by emphasizing key features and suppressing details
  - Models convey structure and help us focus on one thing at a time
- We can use them in systematic testing
  - If a model divides behavior into classes, we probably want to exercise each of those classes!
  - Common model-based testing techniques are based on state machines, decision structures, and grammars
    - but we can apply the same approach to other models

Automated Test Generation using Model Checking

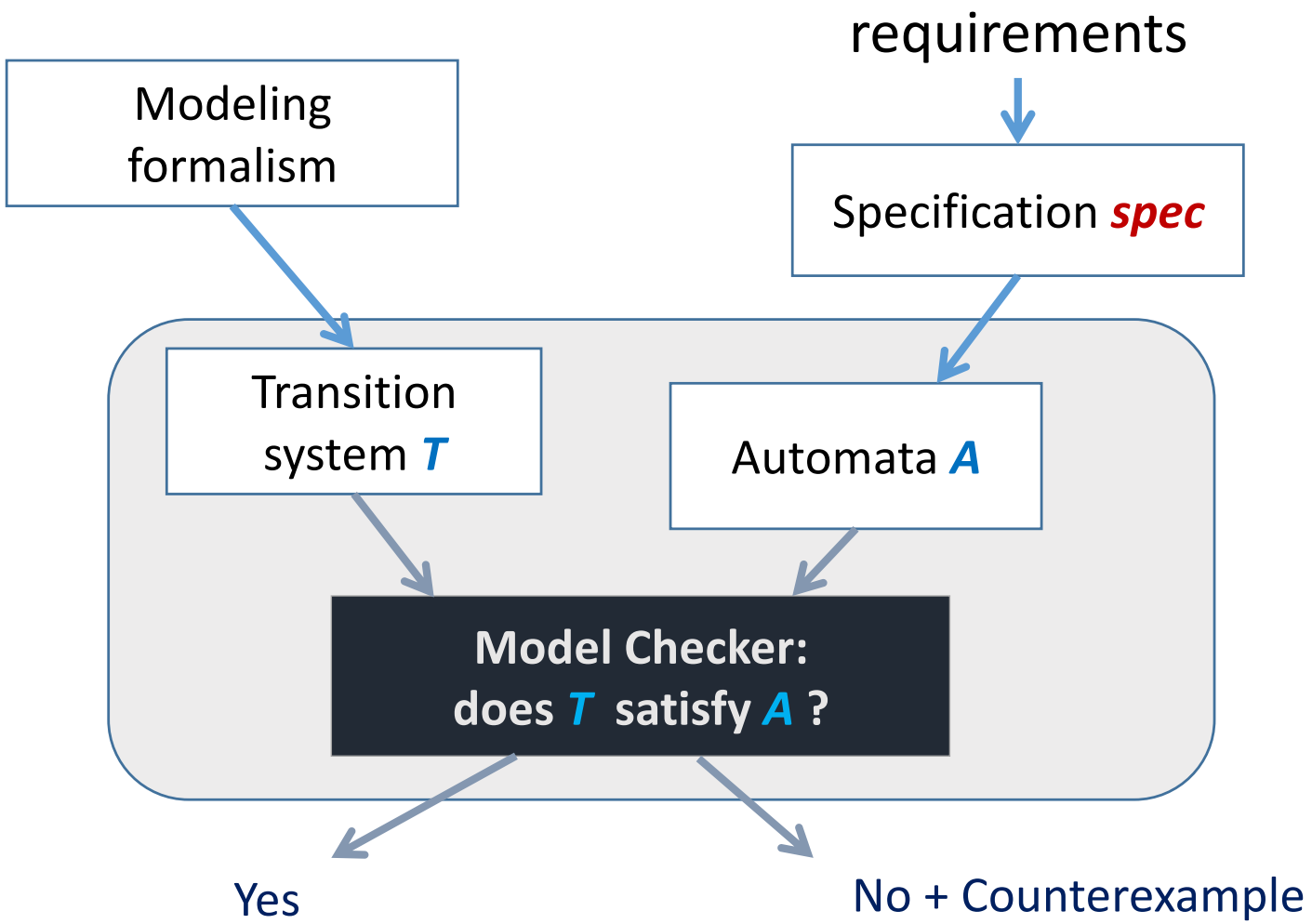
Model-based testing is for automated testing

- But, how to automate the test process?
  - Creation of models is still manual, though domain-specific automation is possible
  - Test specification is also manually determined
- Automation is mainly for the test case generation, test environment generation, and test execution
- Then, how to automate test case generation?
  - Random walk
  - Formal verification

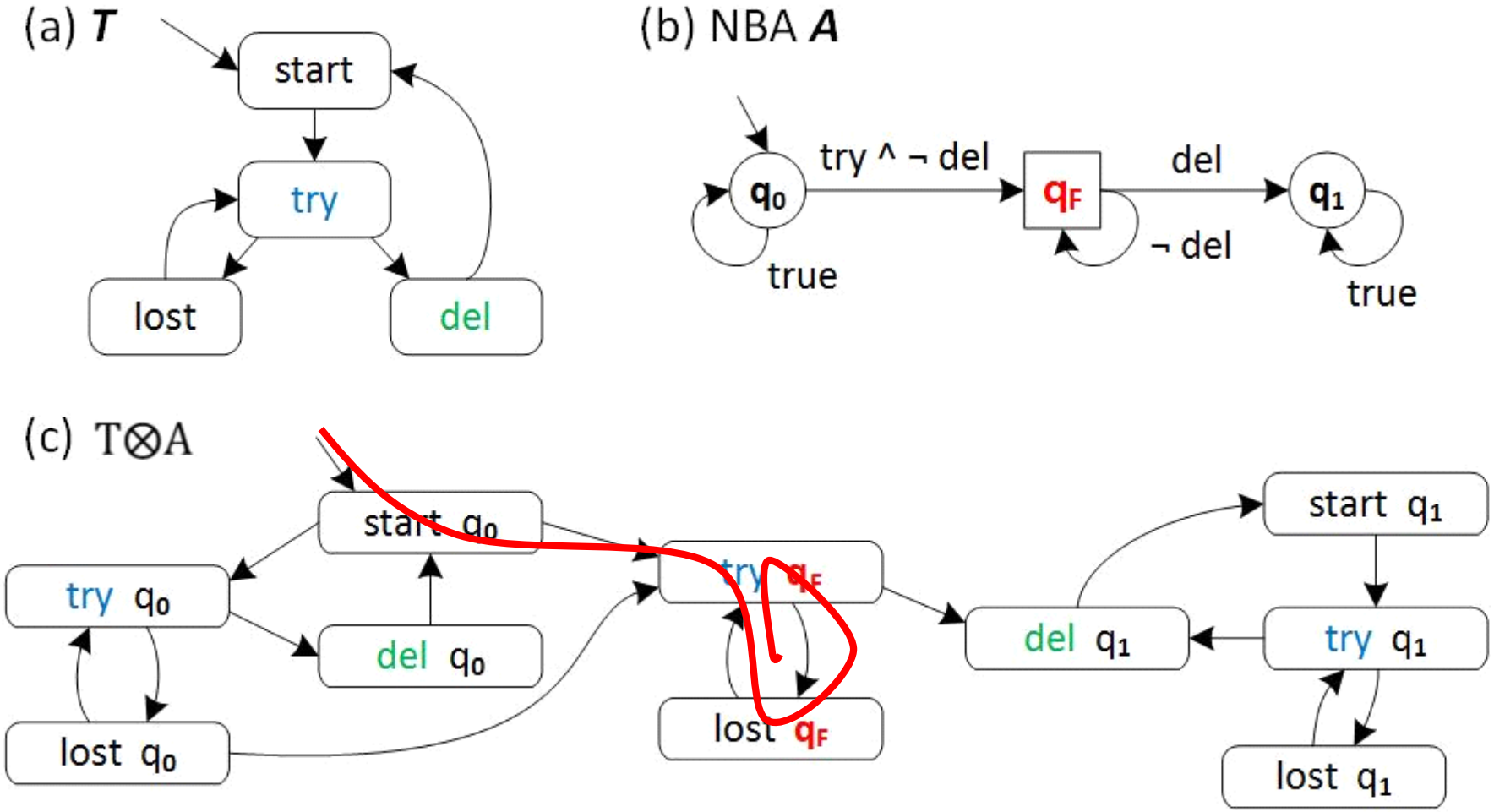
Model-based testing



Techniques used: Model Checking



An example: LTL model checking



Goal of model checking

- Pushing acceleration pedal increases speed
- Rotating the handle x degree changes wheels y degree
- Never opens the doors while driving
- Etc.

p

Formal checking

M ⊨ p

!(M ⊨ p)

Fix this and that ..

Test generation using model checking

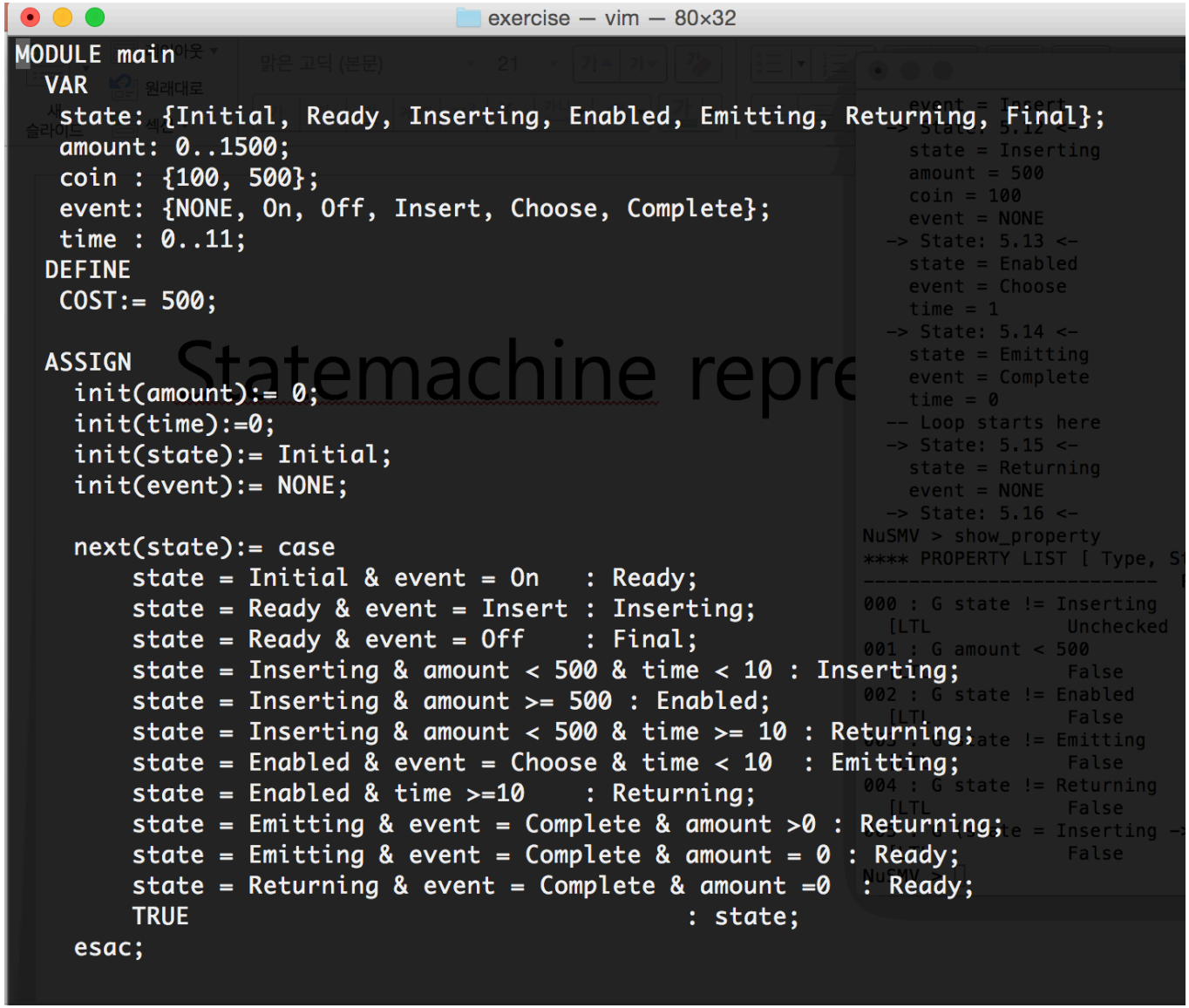
- State coverage
  - For each state s in the model, define a property p as "s is not reachable in the model"
  - Model check M ⊭ p
  - If p is not true, the model checker generates a counterexample which is a execution trace that leads to the state s
- Transition coverage
  - For each transition from s to s', define a property p as "s' cannot be the next state of s"
- Condition coverage
- MC/DC coverage

Hands-on example: NuSMV

- A representative symbolic model checker
  - An extension of SMV from CMU
  - http://nusmv.fbk.eu

Exercise: Chocolate Vending machine re visited

Statemachine representation (Formal)

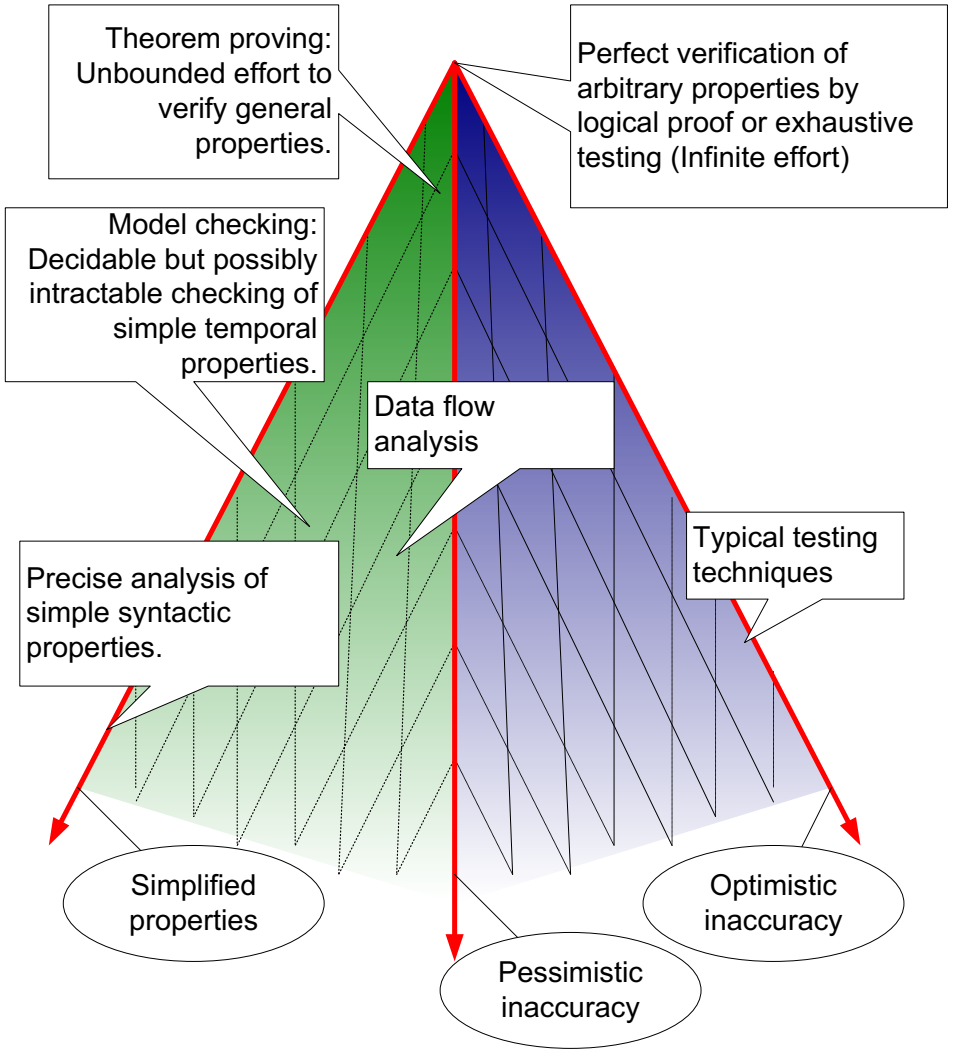


Statemachine representation (Formal)

```
    next(amount):= case
        amount + coin > 1500 : amount;
        state = Initial & event = On    : 0;
        state = Ready & event = Insert : amount+coin ;
        state = Inserting & event =Insert &  time < 10 : amount+coin;
        state = Enabled & event = Choose & time < 10 & amount -COST >=500  : amount - COST;
        state = Returning & event = Complete  : 0;
        TRUE                                  : amount;
    esac;

    next(time):= case
        time +1 > 10   : time;
        state = Initial & event = On    : 0;
        state = Ready & event = Insert : 0;
        state = Inserting & event = Insert & time < 10 : 0;
        state = Inserting & time < 10 : time +1;
        state = Inserting & amount >= 500 : 0;
        state = Enabled & event = Choose & time < 10  : 0;
        state = Enabled &  time < 10  : time+1;
        state = Returning & event = Complete  : 0;
        TRUE                                  : time;
    esac;

LTLSPEC G(state != Inserting)
LTLSPEC G(amount < 500)
LTLSPEC G(state != Enabled)
LTLSPEC G(state != Emitting)
LTLSPEC G(state != Returning)
LTLSPEC G(state = Inserting -> !F state= Returning)
```

Automatic test case generation (using model checking)

Accuracy & Efficiency



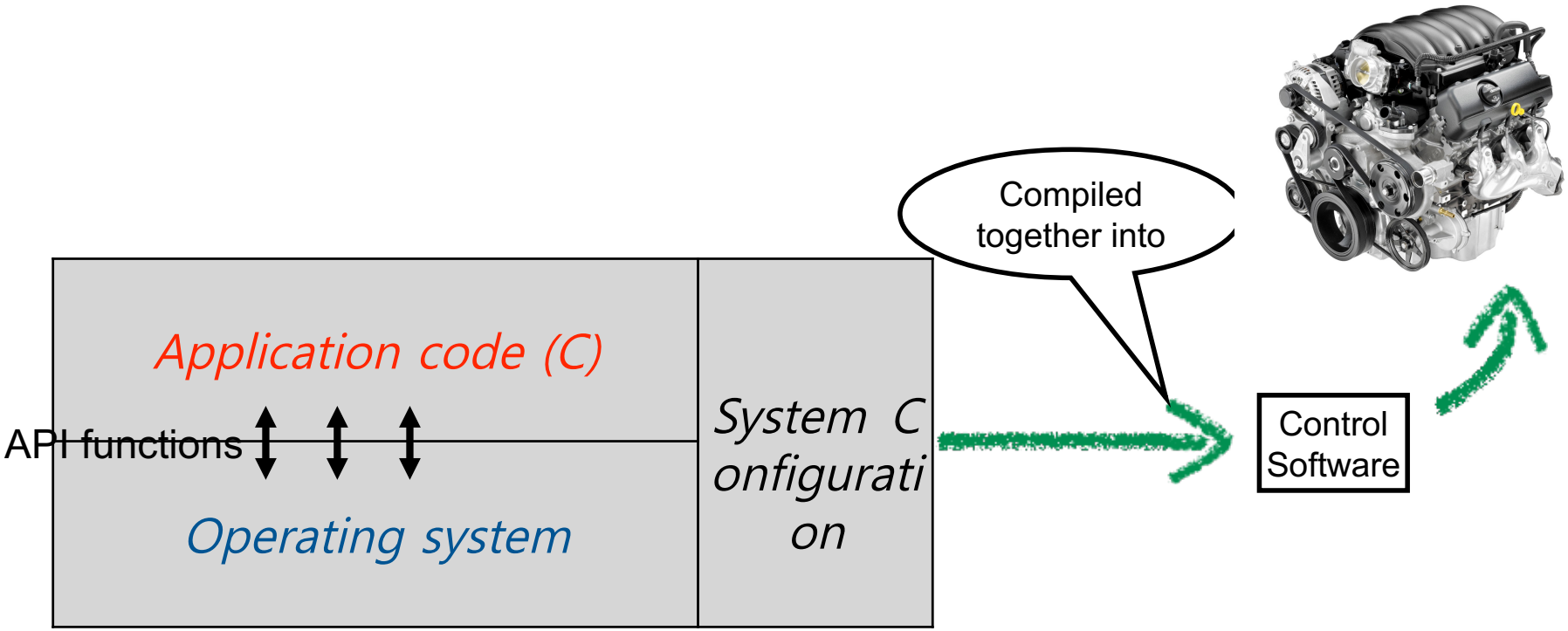
Summary

- When a model is available, automated test generation is not difficult
  - Simplest solution is to use random walk
- The problem is the level of comprehensiveness we would like to achieve
- Formal method is necessary for a comprehensive/guided/foc used test generation

Model-based test generation for embedded operating systems

Real world application

Development of device control software



- The application code is inseparable from operating system kern el

Problem

- A task shall not terminate while occupying resources
- A task that does not own events shall not set events
- A task shall not terminate without calling TerminateTask

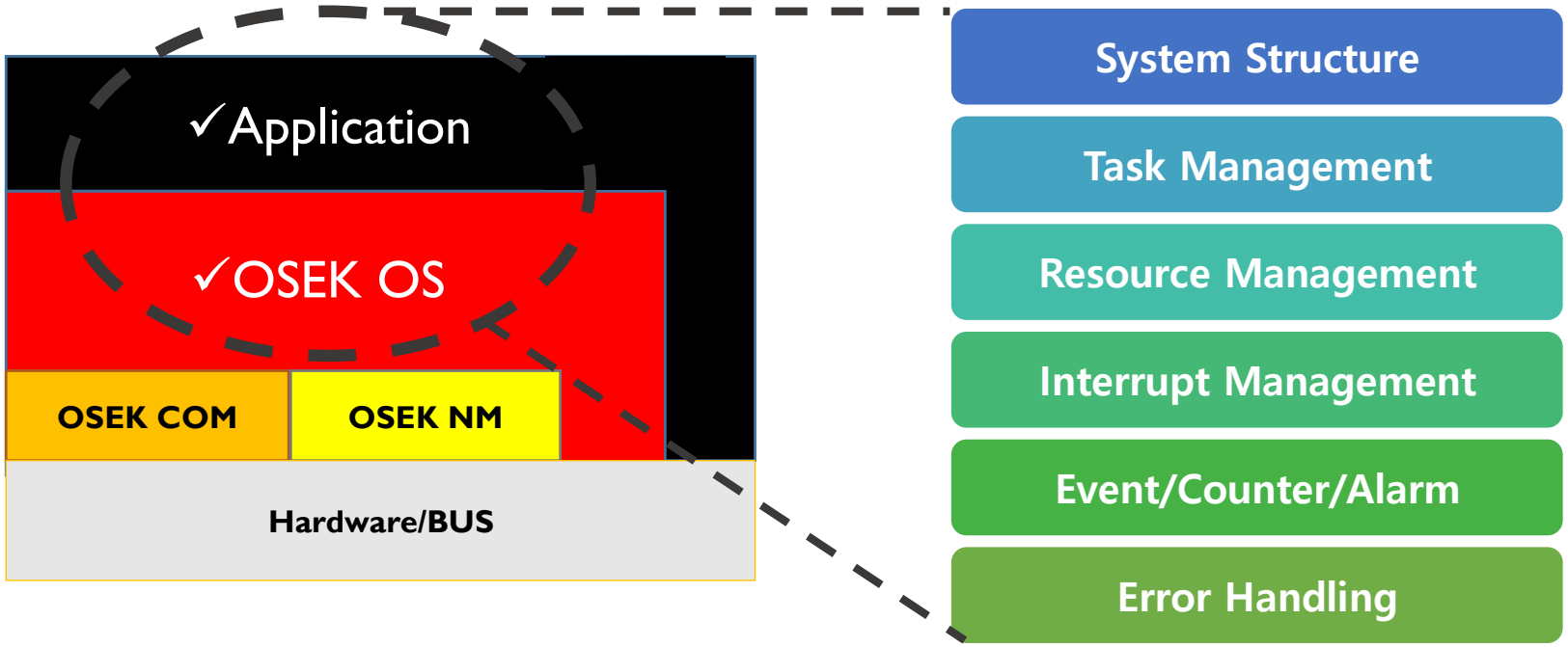
```
01: Task(t1) {
02:   ActivateTask(t2);
03:   SetEvent(t2,e1);
04:   if(condition)
05:     GetResource(r1);
06:   TerminateTask();
07: }
```

Violation of API call constraints may result in system failure

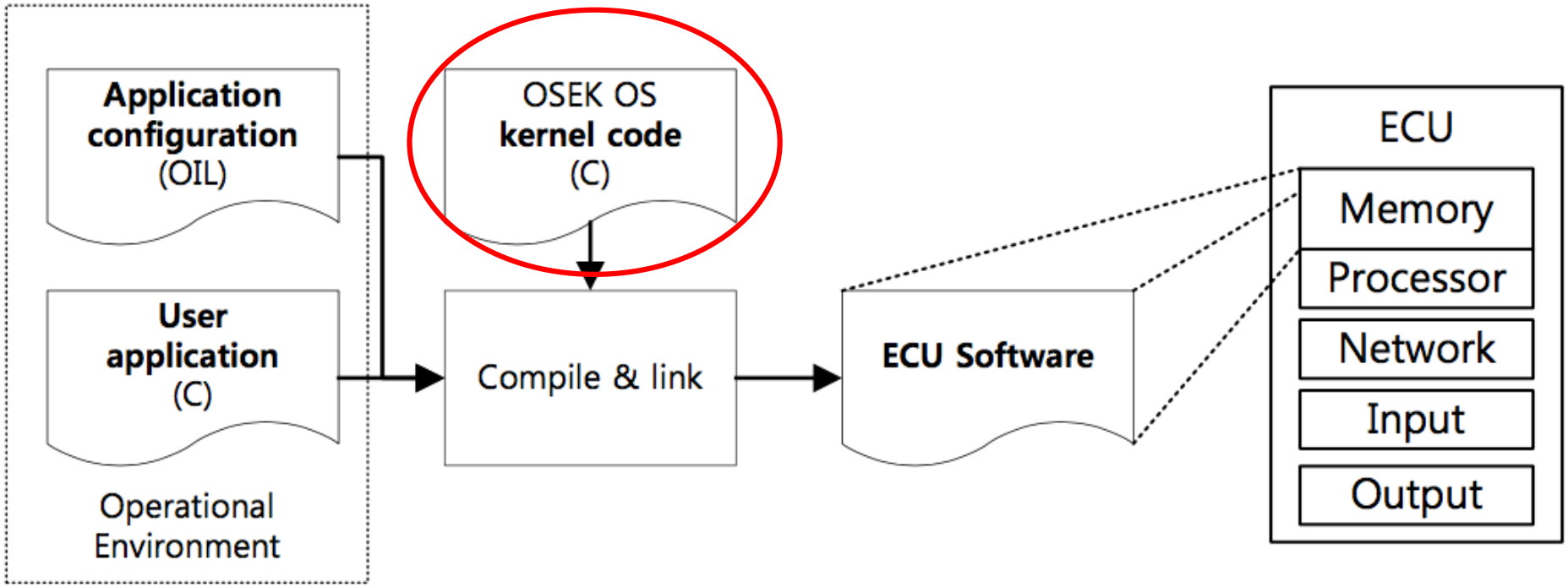
We need to make sure that the OS works safely under unexpected use of API function calls

Target: OSEK/VDX OS

- An international **standard** for distributed control units in vehicles.
- It comprises Communication, Operating System, and Network Management



Issues: Complexity



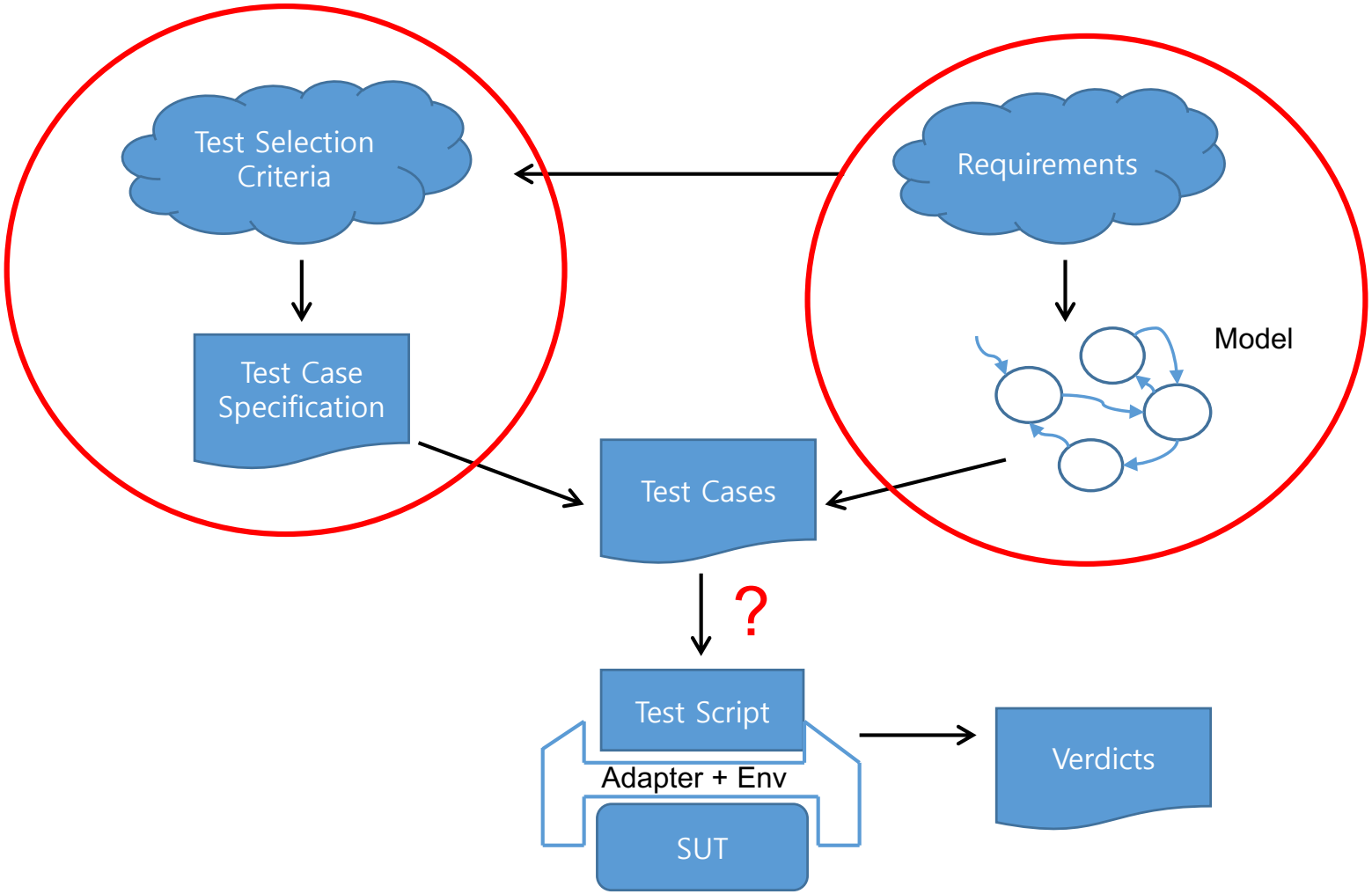
A systematic and efficient verification method is desirable

We solve this problem using

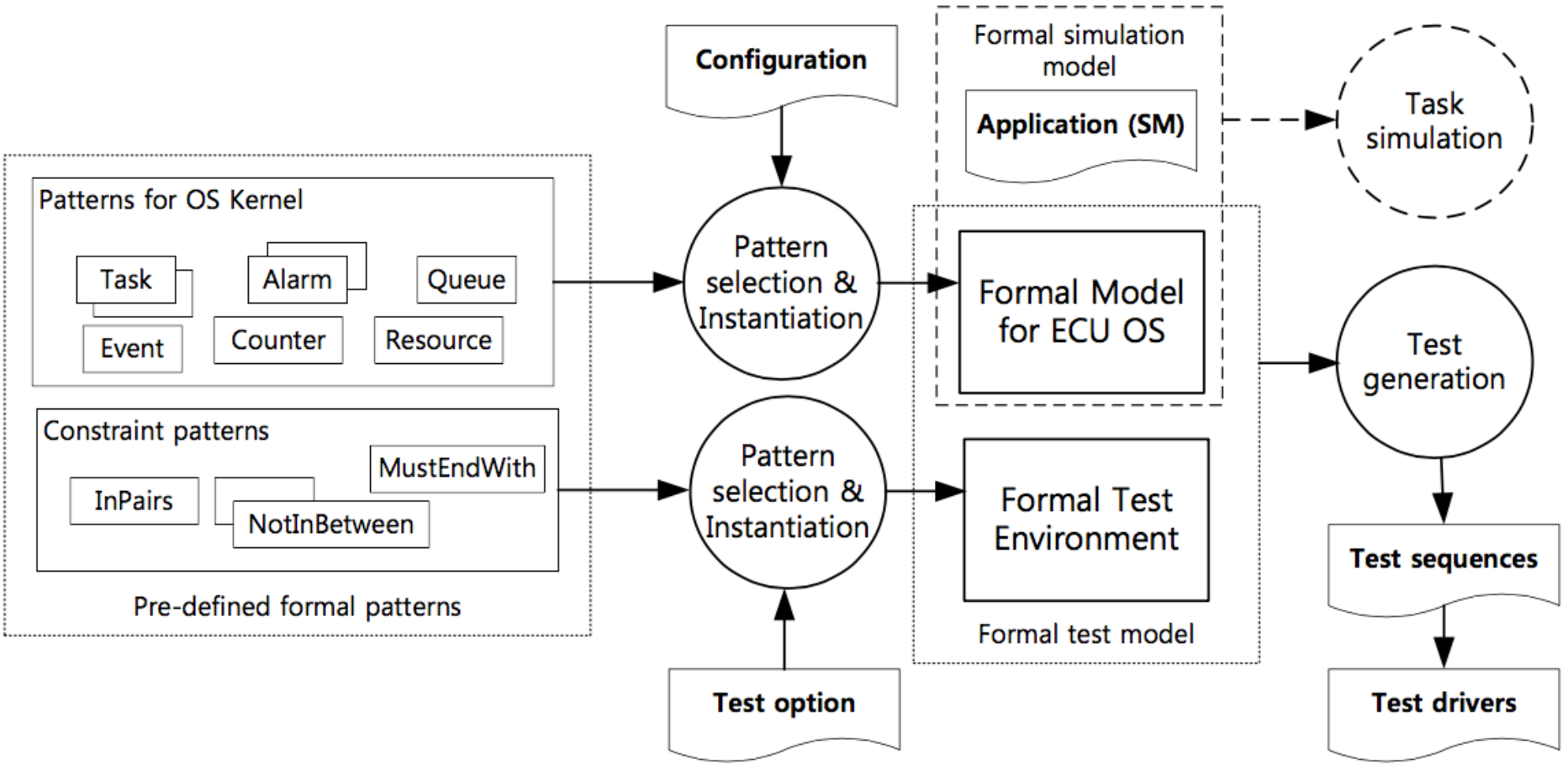
- Domain-specific modeling
- Constraint-based environment modeling
- Modular & pattern-based approach
- Configurable generation of formal models
- Model checking as backend verification engine

- Automate test process as much as possible from model construction
- Reuse models as well as tests

Automating model construction process



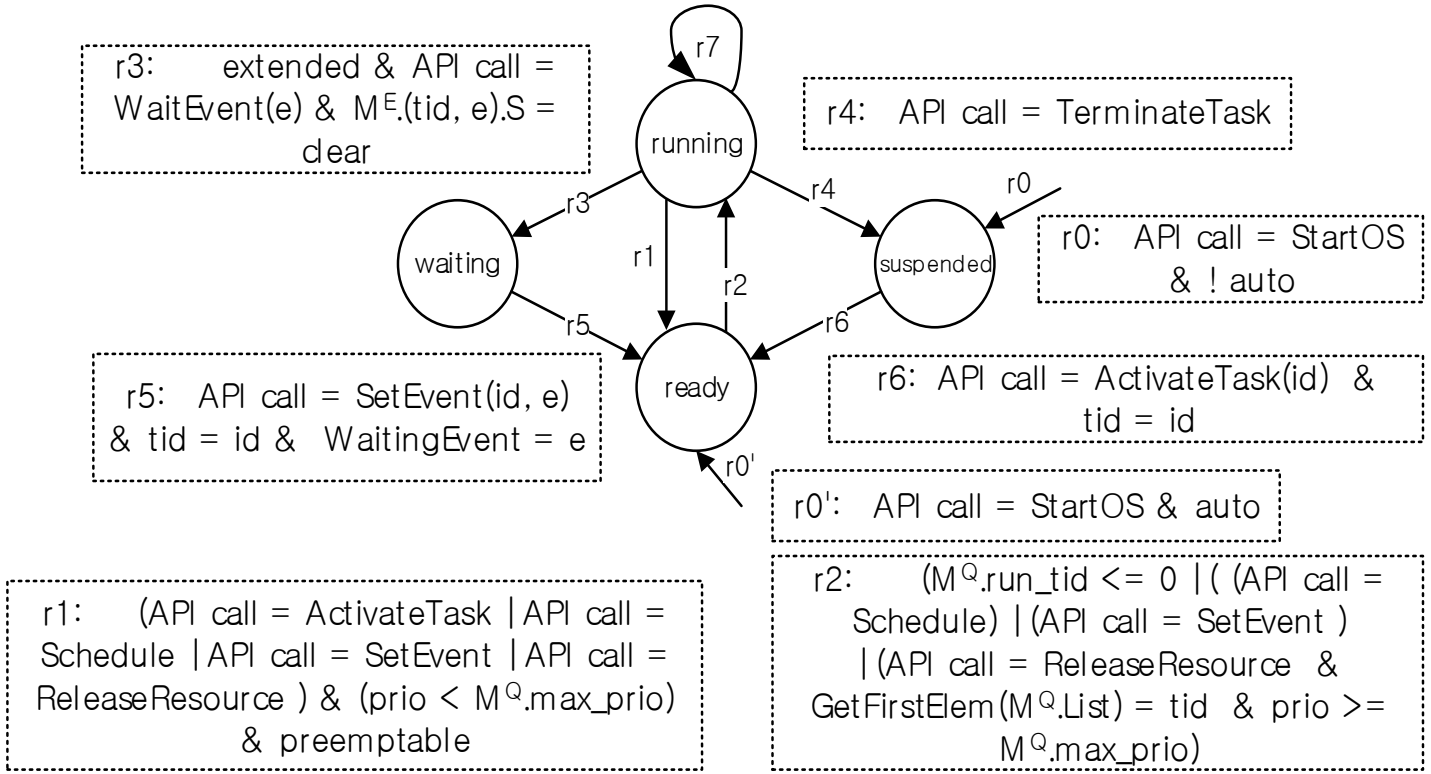
Automating model construction process:
A pattern-based V&V framework



A pattern-based V&V framework

- A set of functional models are pre-defined as parameterized statemachines
  - which is used to auto-generate a formal OS model for each configur ation
- A set of constraint patterns are pre-defined as parameterized statemachines
  - They represent operational constraints of underlying operating syste ms
  - Test engineers can choose a subset of the constraints to impose con straint on the operational environment and generate test sequences
- It supports
  - automated safety checking for operating systems
  - automated API-call constraint checking for control software
  - simulation of task sequences and validation of task design

Formal specifications: Tasks




```
MODULE Task(env, tid, ptiv, pri, autostrt, extended, rq, e_run, res, evt)
```

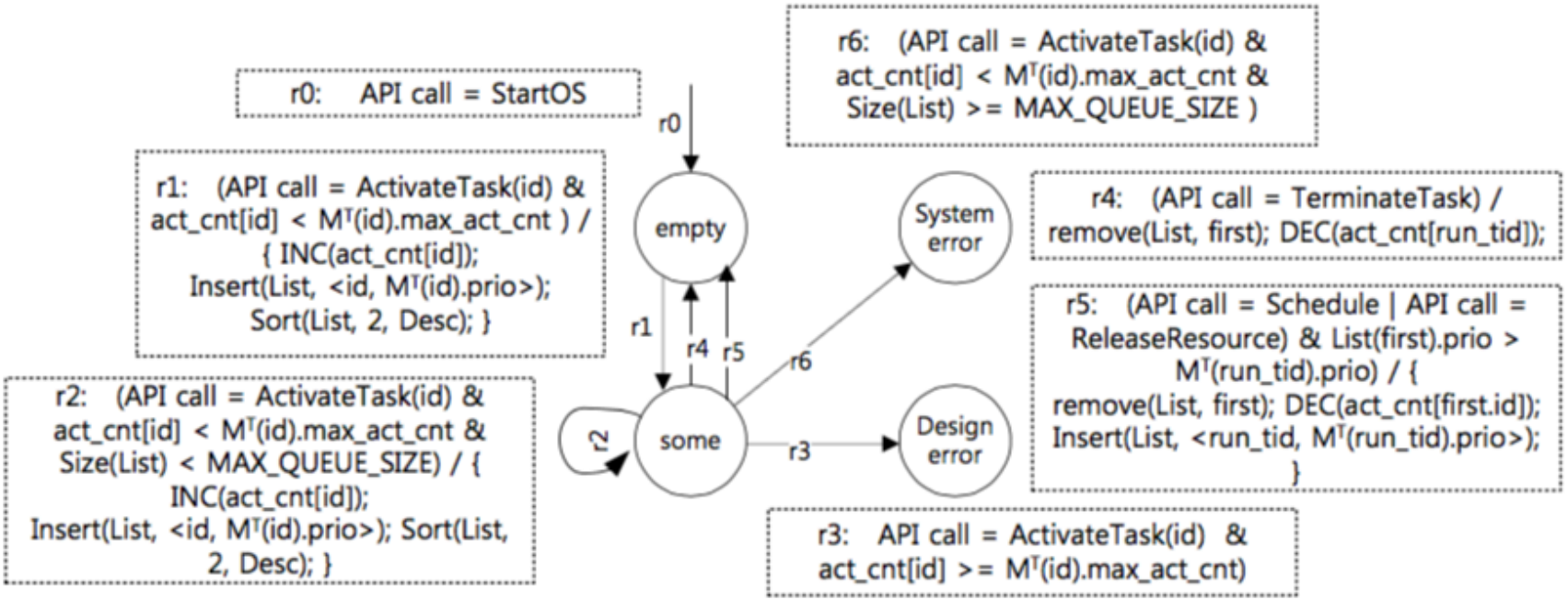
Formal specifications: Tasks

```
MODULE Task(env, tid, ptiv, pri, autostrt, extended, rq, e_run, res, evt)
```

```
MODULE Task(env, tid, ptiv, pri, autostrt, extended, rq, e_run, res, evt)
VAR
state : {SUS, RDY, WIT, RUN};
prio : 1..5;
go_rdy : boolean;
wait_e : 1..2;
c2 : NotInBetween(env,
c3 : MustEndWith(env,

FROZENVAR
id : 1..5;
autostart : boolean;

DEFINE
exc_set2 := {TT, WE};
end_set3 := {TT};
ext := extended;

ASSIGN
init(autostart) := aut
init(id)        := tid
init(state)     := cas
          autostrt : RDY
          TRUE : SUS;
esac;
init(wait_e)    := 1;
next(wait_e)    := case
          env.nWE & state = RUN & nex    tate) = WIT : next(env.p_e);
          TRUE : wait_e;
esac;
next(state)     := case
          state = RDY : case
          !e_run & prio >= rq.max_prio : RUN;
          (env.nSC | env.nRR | env.nSE) & rq.pq[prio][0] = 1 & prio > rq.max_prio : RUN;
          TRUE : state;
esac;
          state = RUN : case
          next(env.api) = None & ptiv = TRUE & prio < rq.max_prio : RDY;
          env.nAT & ptiv = TRUE & prio < (rq.max_prio) : RDY;
          (env.nSC | env.nSE | env.nRR) & prio < rq.max_prio & ptiv = TRUE : RDY;
          env.nCT : SUS;
          env.nTT : SUS;
          ext & env.nWE & next(evt.valid) & evt.state[tid][next(env.p_e)] = clear : WIT;
          TRUE : state;
esac;
          state = SUS & (env.nAT | env.nCT) & next(env.p_t) = id : {state, RDY};
          state = WIT & env.nSE & next(env.p_t) = tid & evt.valid & wait_e = next(env.p_e) : RDY;
          TRUE : state;
esac;
init(go_rdy)    := (state = RDY);
next(go_rdy)    := case
          state = SUS & next(state) = RDY : TRUE;
          TRUE : FALSE;
esac;
init(prio)      := pri;
next(prio)      := case
          env.nGR & state = RUN & next(res.ceil_prio) > prio : next(res.ceil_prio);
          env.nRR : pri;
          TRUE : prio;
esac;
```

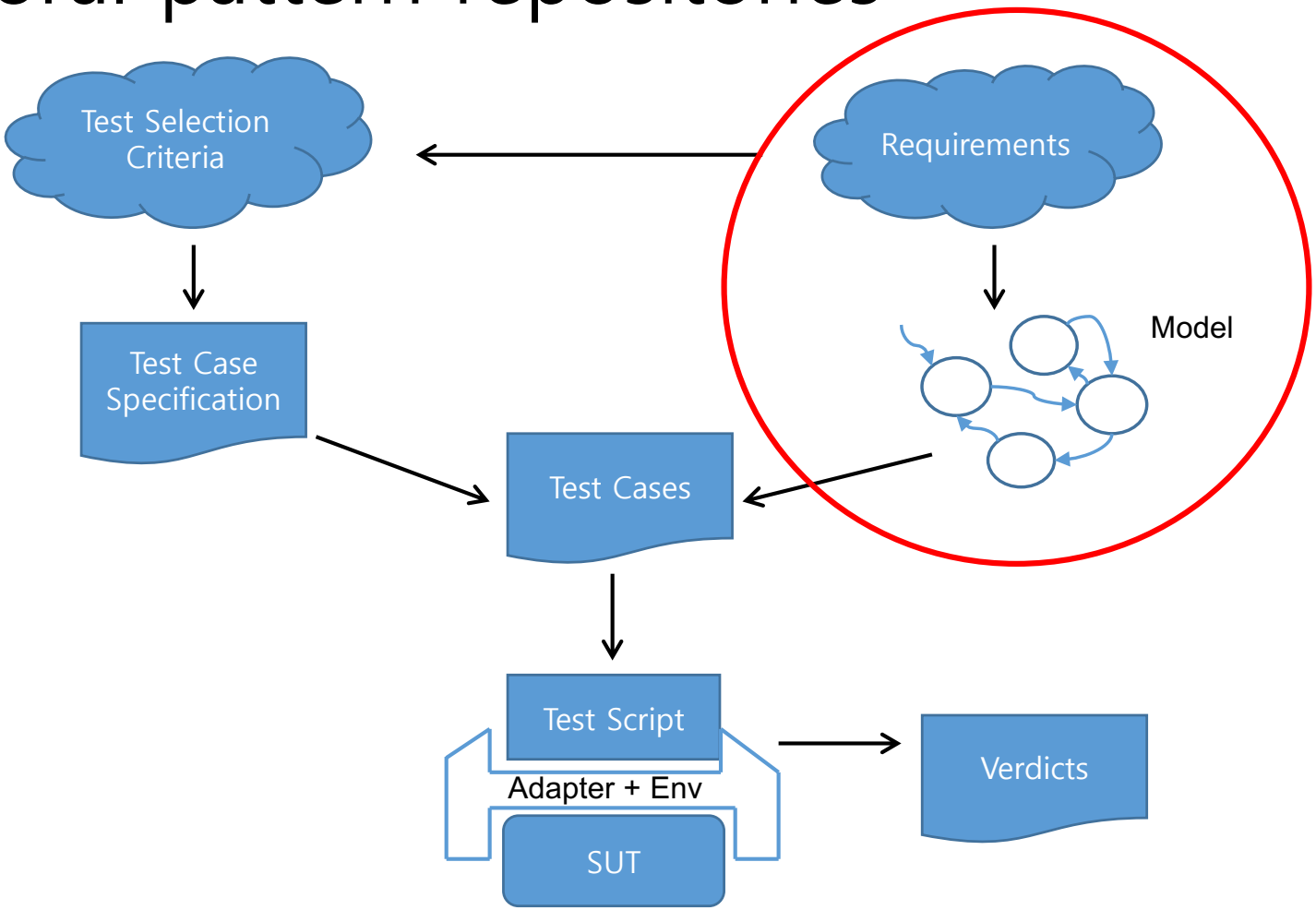
```
next(state) := case
    state = RDY :
      case
        !e_run & prio >= rq.max_prio : RUN;
        (env.nSC | env.nRR | env.nSE) & rq.pq[prio][0] = 1 & prio > rq.max_prio : RUN;
        TRUE : state;
      esac;
    ....
```

Formal specifications: Scheduler

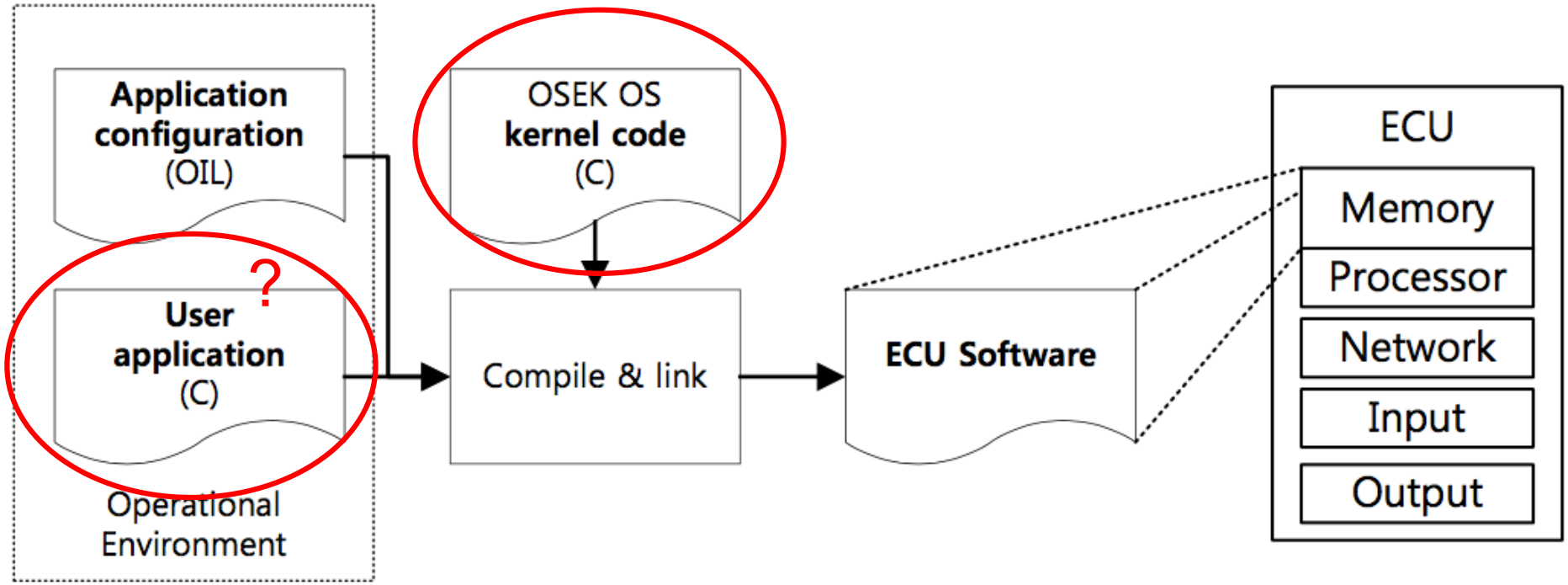



```
MODULE Scheduler(tasks)
```

Configurable model generation using be havioral pattern repositories



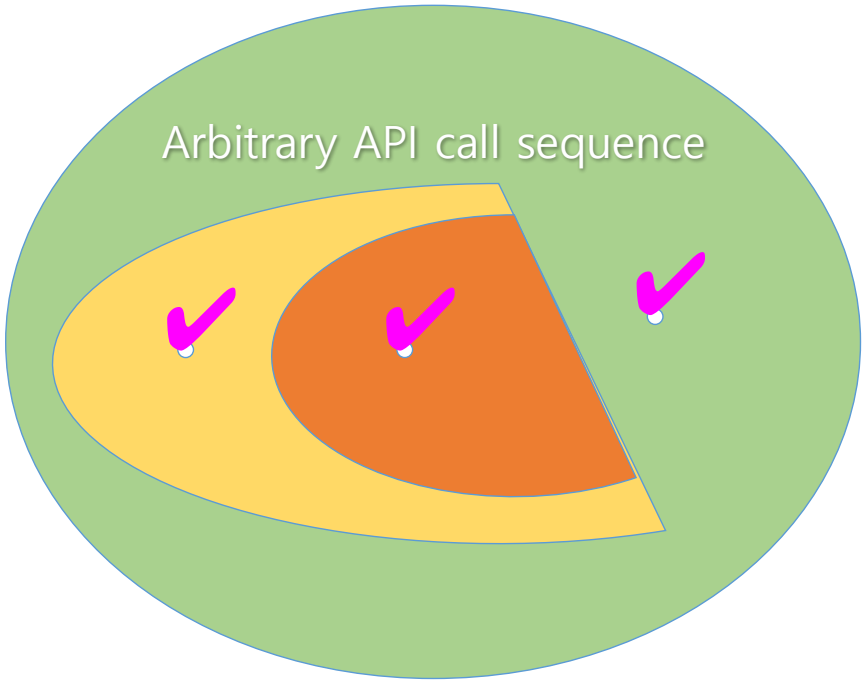
Modeling operational environment using constraint patterns



A systematic and efficient verification method is desirable

Constraint-based Environment Modeling

- Systematically apply constraints to the universal environ ment



Examples of Constraints

1. Ending a task without a call to TerminateTask or ChainTask is strictly forbidden and causes undefined behavior
2. TerminateTask, ChainTask, Schedule, WaitEvent shall not be called while a resource is occupied
3. A task calling WaitEvent shall go to the waiting state and shall not be called while a resource is occupied
4. OSEK strictly forbids nested access to the same resource
5. A task shall not terminate without releasing resources

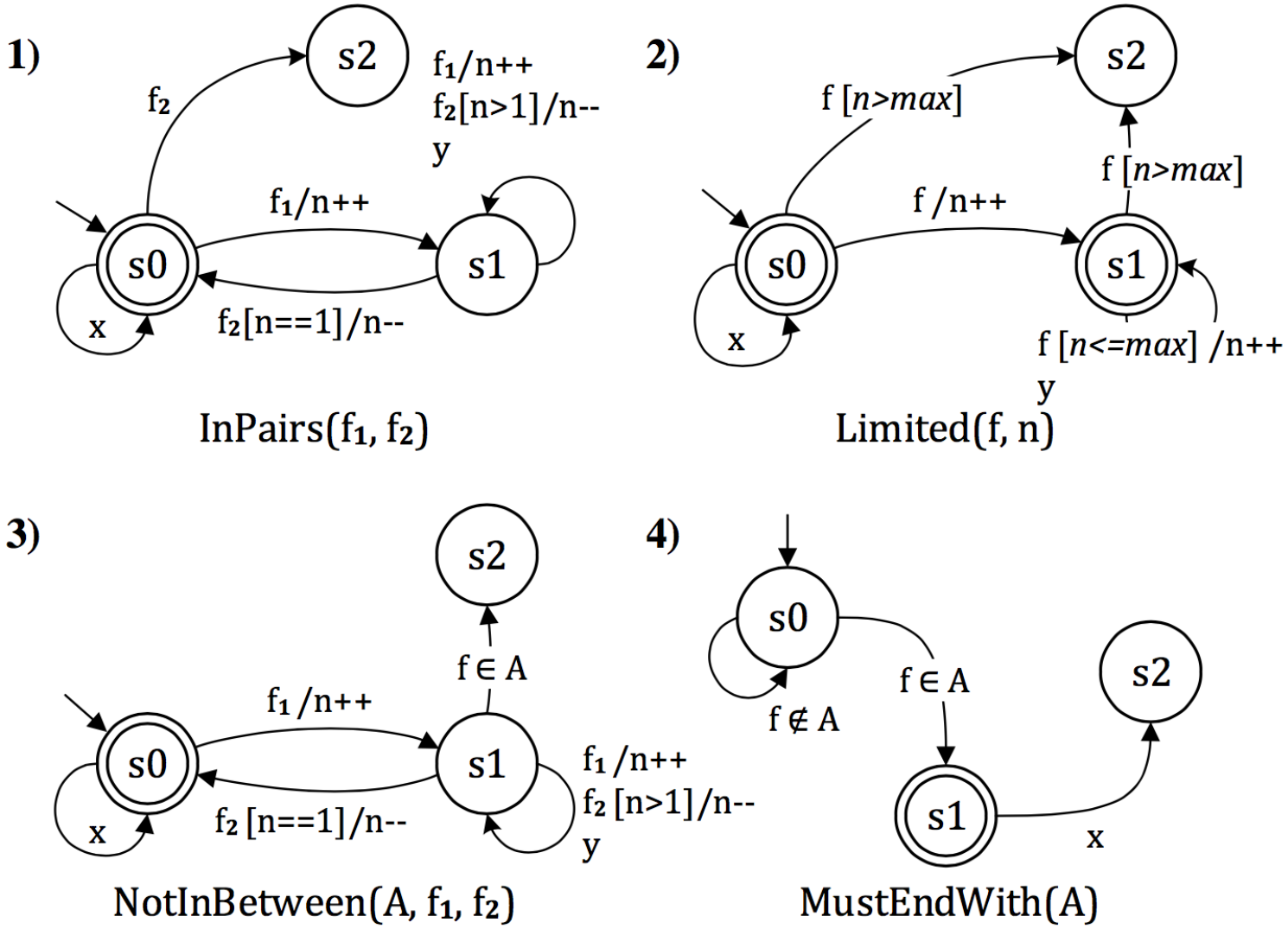
Formal Constraint Specification

< OSEK_CSL>

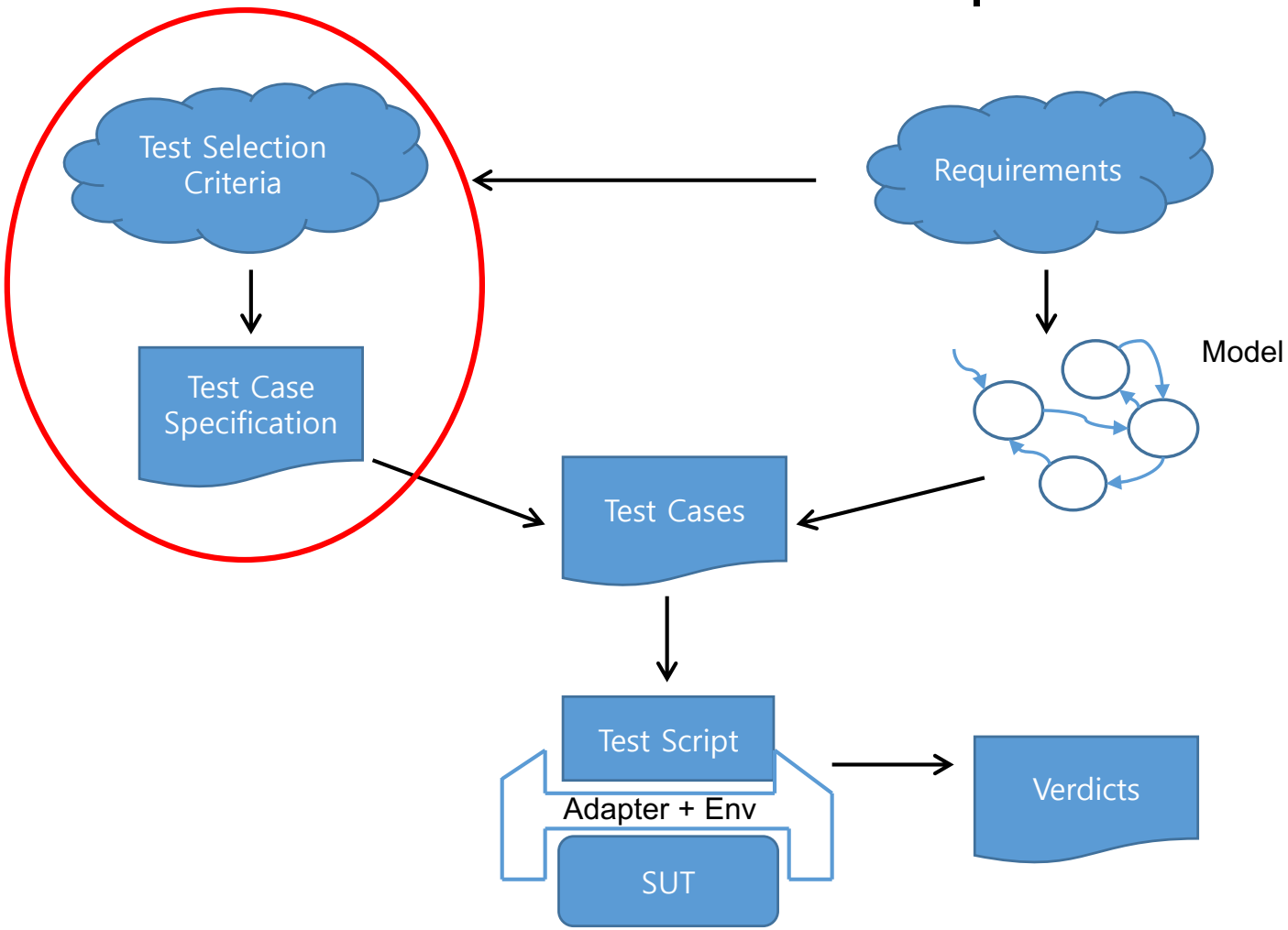
1. **InPairs(f1, f2) :** A system call f1 shall be followed by f2
2. **Limited(f, n)/ SetLimited(A, n) :** The number of calls to f is li mited by n
3. **NotInBetween(f, f1, f2) :** A system call f shall not be called in bet ween two system calls f1 and f2
4. **MustEndWith(f) :** No system call shall be made after a call to f

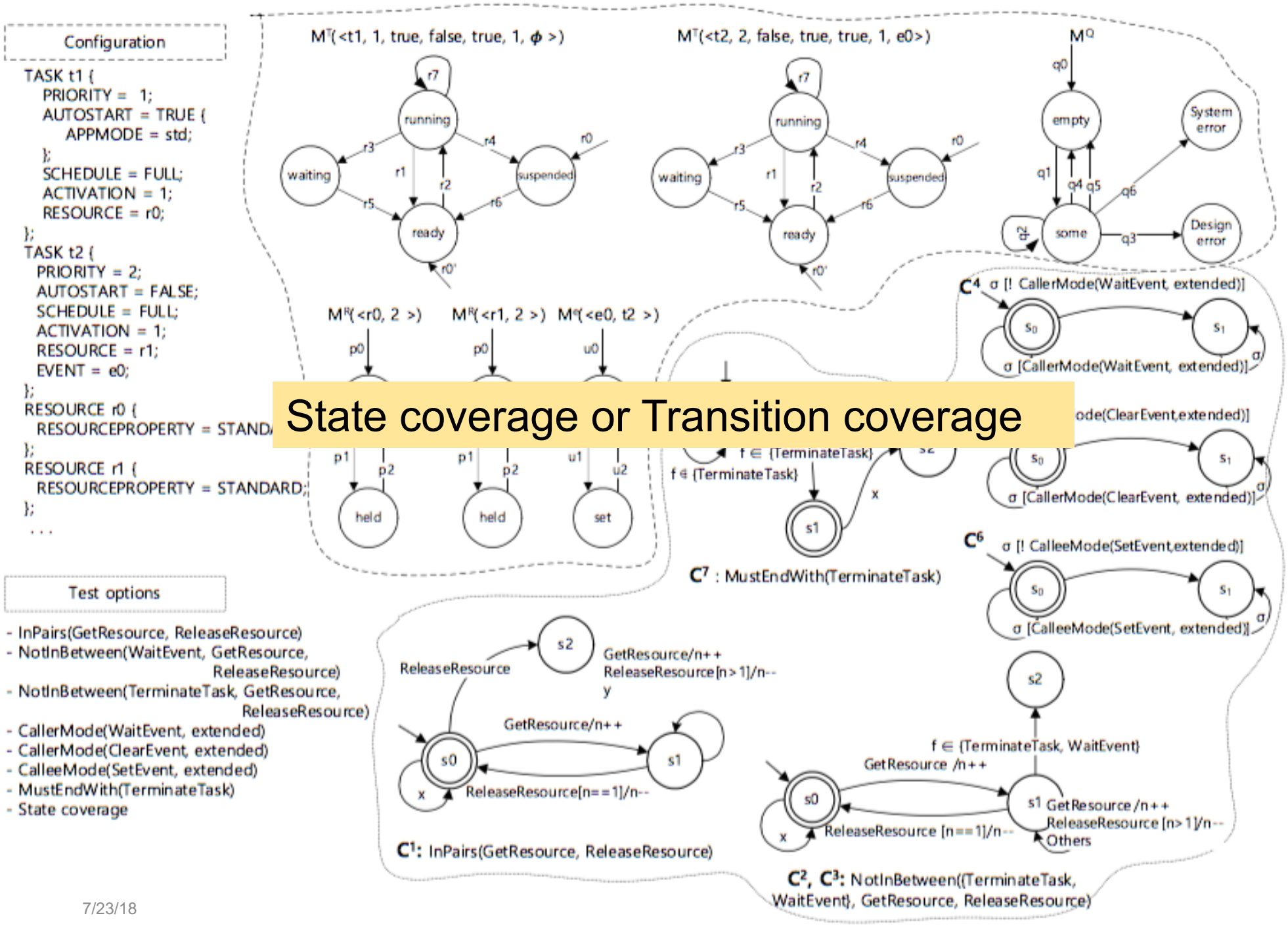
$$Limited(f,n) = \{w \mid w \in \Sigma^*,\ n_f(w) \leq n\}$$
$$SetLimited(A,n) = \{w \mid w \in \Sigma^*,\ SUM_{f_i \in A}(n_{f_i}(w)) \leq n\}$$
$$MustEndWith(A) = \{wf \mid w \in (\Sigma - A)^*,\ f \in A\}$$

Constraint Patterns



Constraint-based test case specification




Configuration
TASK t1 {
PRIORITY = 1;
AUTOSTART = TRUE {
APPMODE = std;
};
SCHEDULE = FULL;
ACTIVATION = 1;
RESOURCE = r0;
};
TASK t2 {
PRIORITY = 2;
AUTOSTART = FALSE;
SCHEDULE = FULL;
ACTIVATION = 1;
RESOURCE = r1;
EVENT = e0;
};
RESOURCE r0 {
RESOURCEPROPERTY = STANDA
};
RESOURCE r1 {
RESOURCEPROPERTY = STANDARD;
};
. . .
Test options
- InPairs(GetResource, ReleaseResource)
- NotInBetween(WaitEvent, GetResource, ReleaseResource)
- NotInBetween(TerminateTask, GetResource, ReleaseResource)
- CallerMode(WaitEvent, extended)
- CallerMode(ClearEvent, extended)
- CalleeMode(SetEvent, extended)
- MustEndWith(TerminateTask)
- State coverage
M^T(<t1, 1, true, false, true, 1, ϕ >)
M^T(<t2, 2, false, true, true, 1, e0>)
M^Q
running
waiting
suspended
ready
empty
some
System error
Design error
M^R(<r0, 2 >)
M^R(<r1, 2 >)
M^e(<e0, t2 >)
held
held
set
State coverage or Transition coverage
C4 σ [! CallerMode(WaitEvent, extended)]
σ [CallerMode(WaitEvent, extended)]
σ [CallerMode(ClearEvent, extended)]
f ∈ {TerminateTask}
f ∉ {TerminateTask}
C7 : MustEndWith(TerminateTask)
C6 σ [! CalleeMode(SetEvent,extended)]
σ [CalleeMode(SetEvent, extended)]
ReleaseResource
GetResource/n++
ReleaseResource[n>1]/n--
GetResource/n++
ReleaseResource[n==1]/n--
C1: InPairs(GetResource, ReleaseResource)
f ∈ {TerminateTask, WaitEvent}
GetResource /n++
ReleaseResource [n==1]/n--
GetResource /n++
ReleaseResource [n>1]/n--
Others
C2, C3: NotInBetween({TerminateTask, WaitEvent}, GetResource, ReleaseResource)

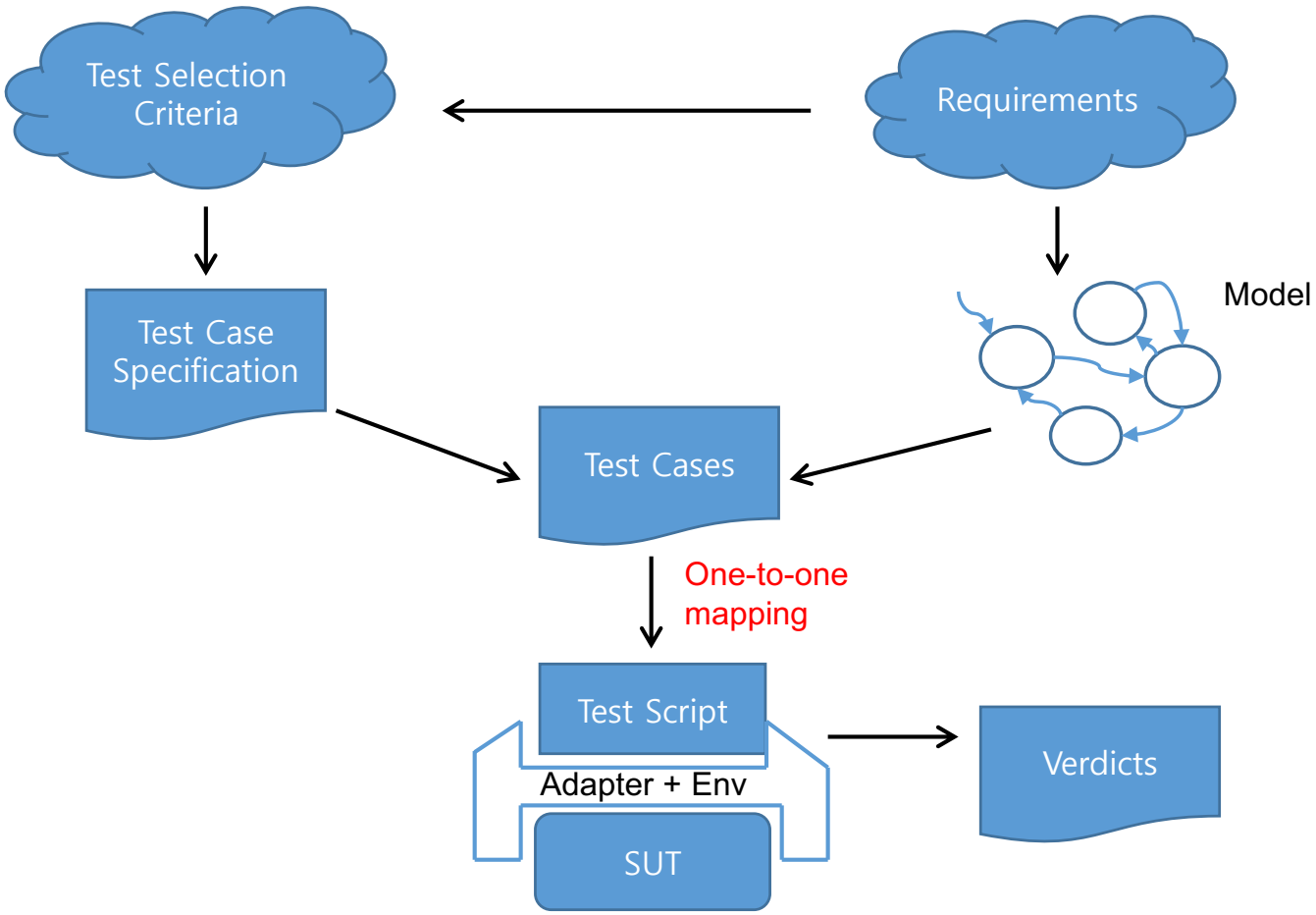
Trap properties

- A trap property is for generating counterexamples
  - e.g., a certain state cannot be reached from the initial state
- Typically a false property
- Model checker can generate an execution trace showing how the pr operty can be violated
  - The counterexample trace becomes a test sequence

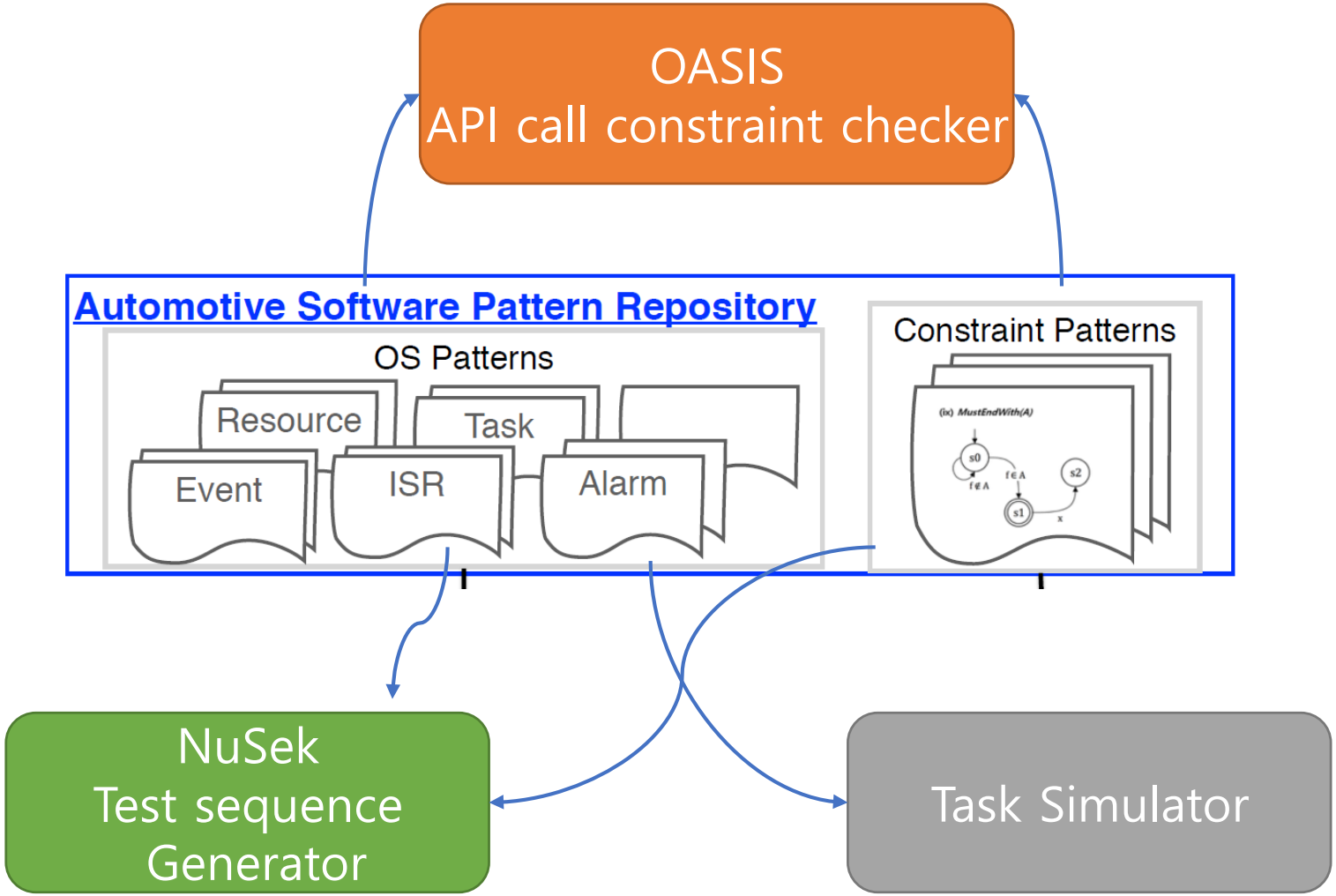
Definition 8 *(trap properties) Suppose there are n number of constraints specified in OSEK_CSL, which are formalized in finite automata* $A = \{A^i\}$, $i = 1 \ldots n$, *where* $A^i = (S^i, \Sigma, \Gamma, \delta^i, s_0^i, z^i, \{s_f^i\})$. *Let* I *be the set of indices of the elements in* A.

1. $tp^C(1) \stackrel{\text{def}}{=} G((\prod_{i \in I} S^i = s_0^i) \rightarrow ! \; F(\prod_{i \in I} S^i = s_f^i))$ *(for a perfect environment model).*
2. $tp^C(\frac{k}{n}) \stackrel{\text{def}}{=} G((\prod_{i \in I} S^i = s_0^i) \rightarrow ! \; F(\prod_{i \in K} S^i = s_f^i))$, *where* $0 < k < n$ *and* K *is the set of indices of the elements in* B $(\emptyset \neq B \subset A)$ *with its cardinality* k *(for erroneous environment models).*
3. $tp^C(0) \stackrel{\text{def}}{=} G((\prod_{i \in I} S^i = s_0^i) \rightarrow ! \; F(\prod_{i \in I} S^i \neq s_f^i))$ *(for false environments).*

Test driver generation



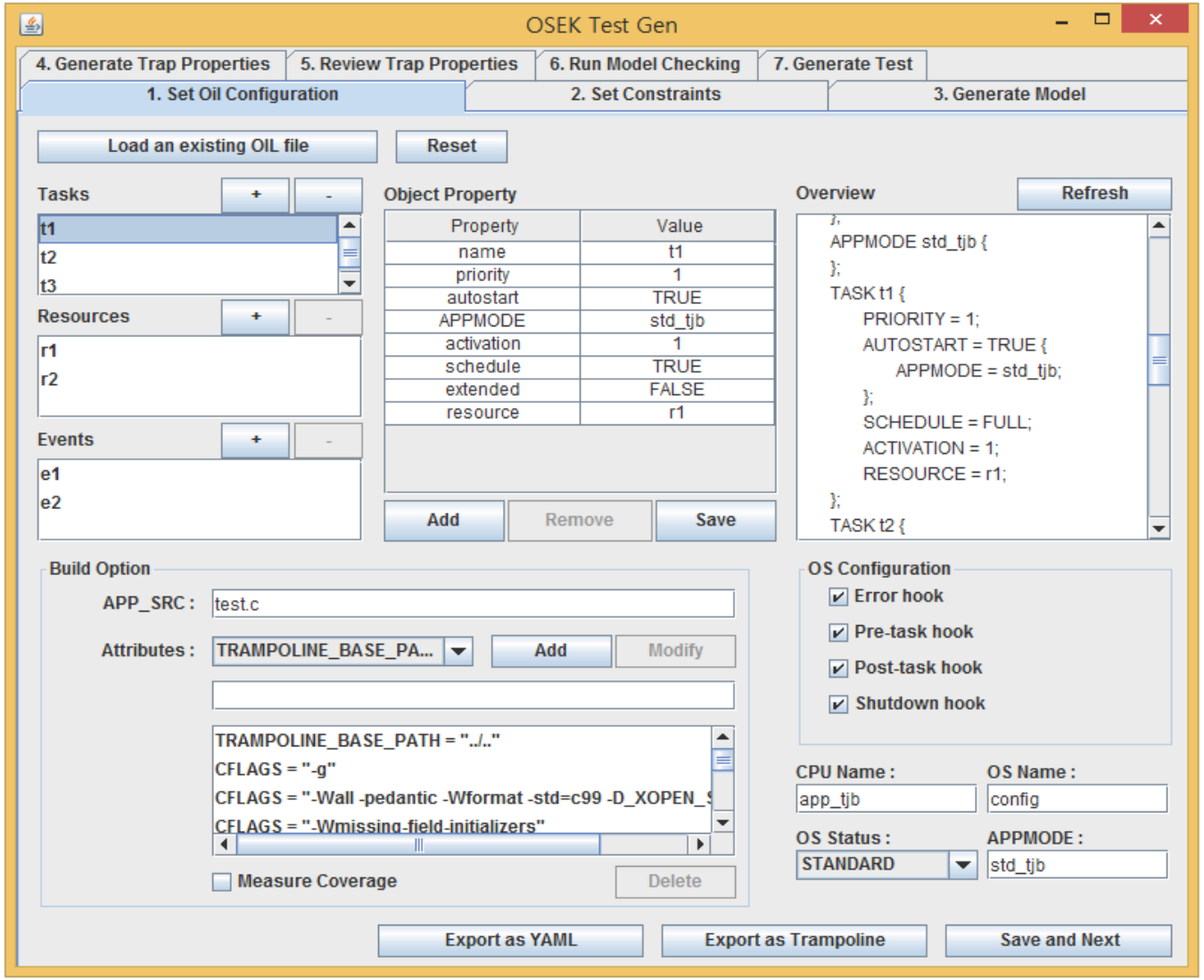
Prototype tools



Prototype tools (NuSek)

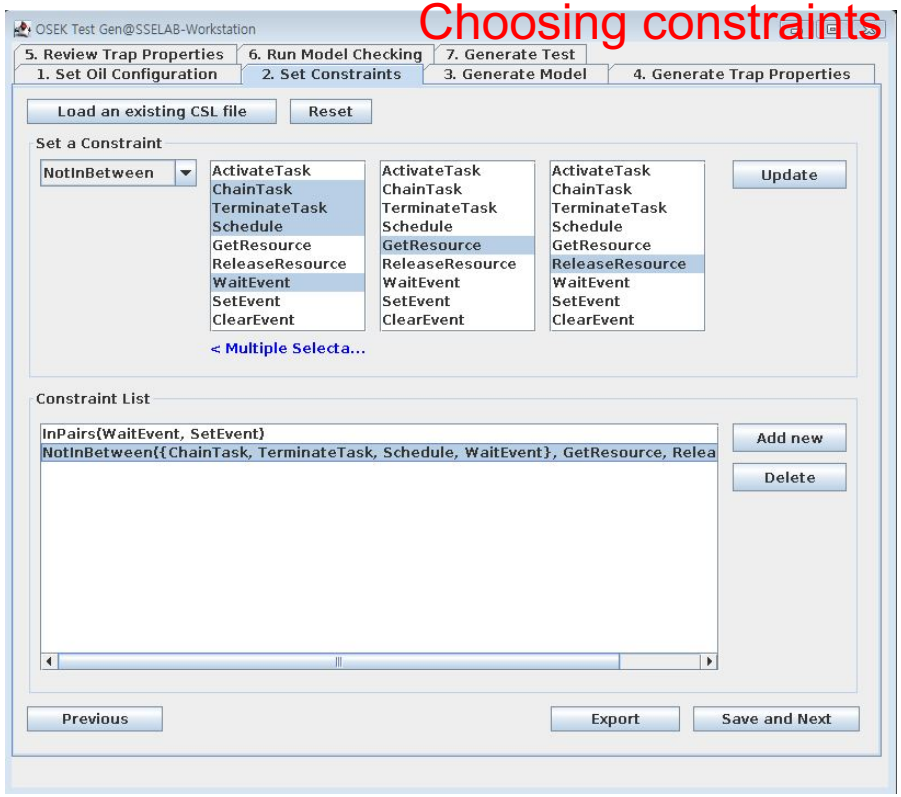
Tool demo available at https://youtu.be/ajhV9N2jCb8

Setting system configuration



Tool demo available at https://youtu.be/ajhV9N2jCb8

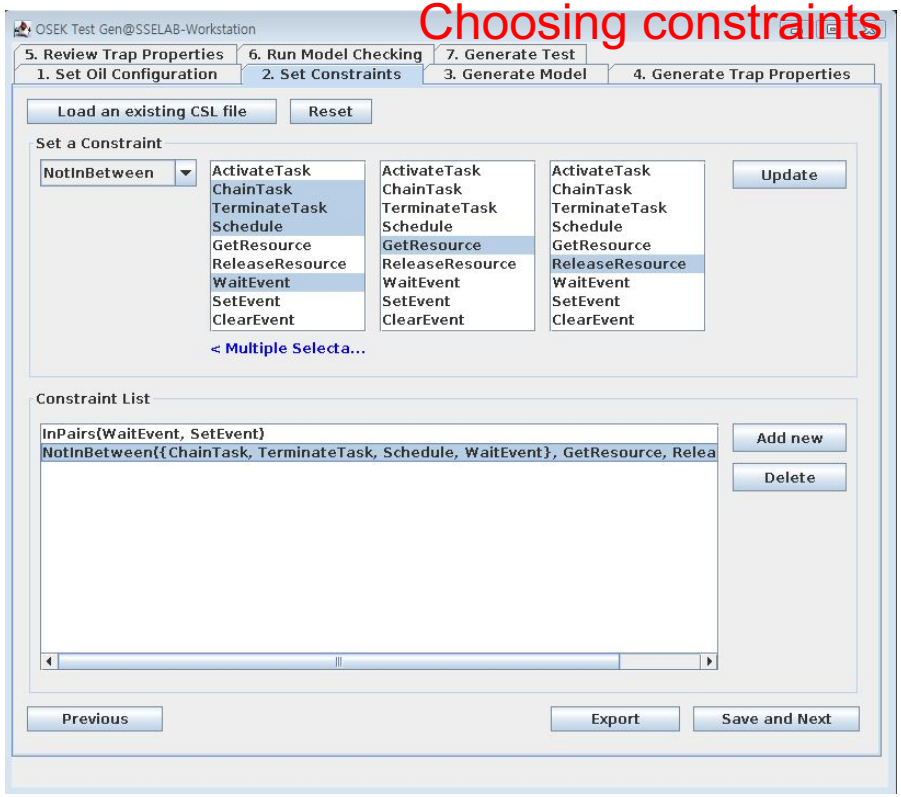
Prototype tools



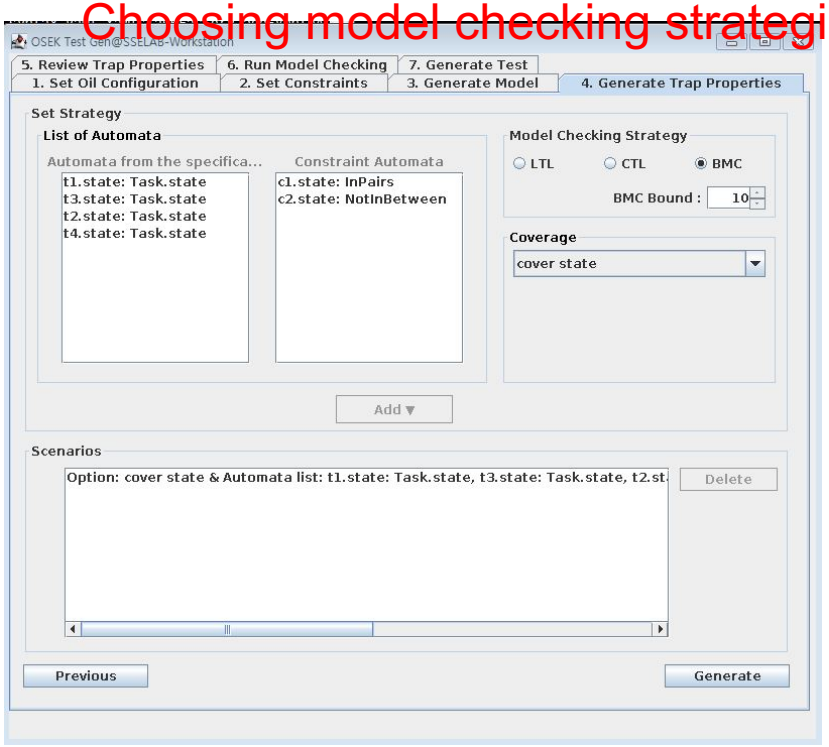
Tool demo available at https://youtu.be/ajhV9N2jCb8

Prototype tools

Choosing constraints



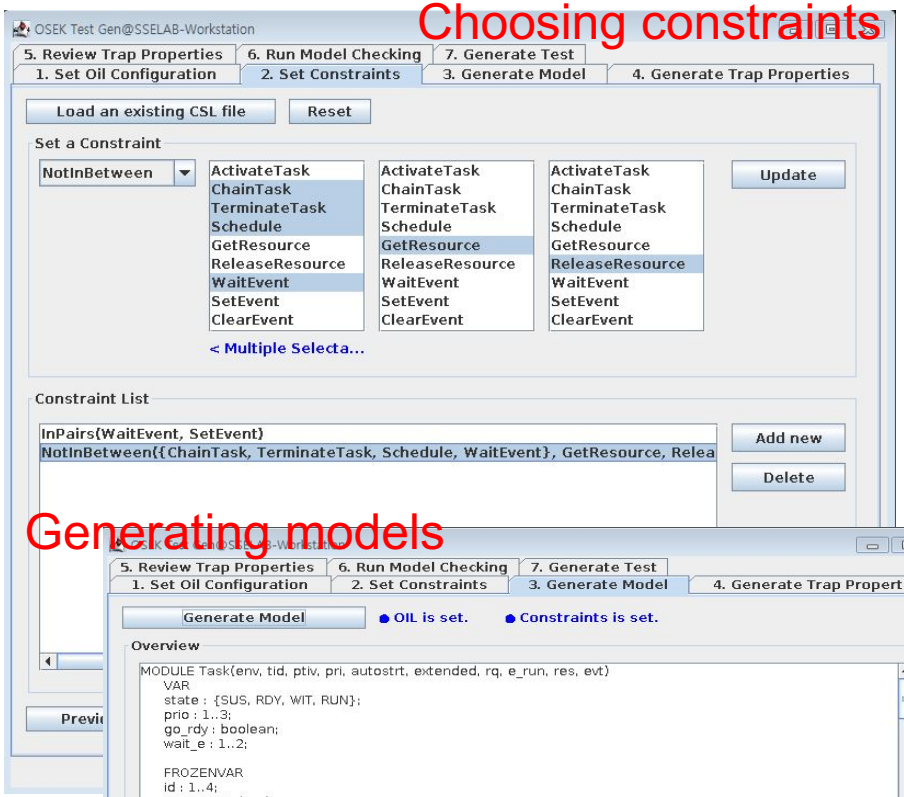
Choosing model checking strategies



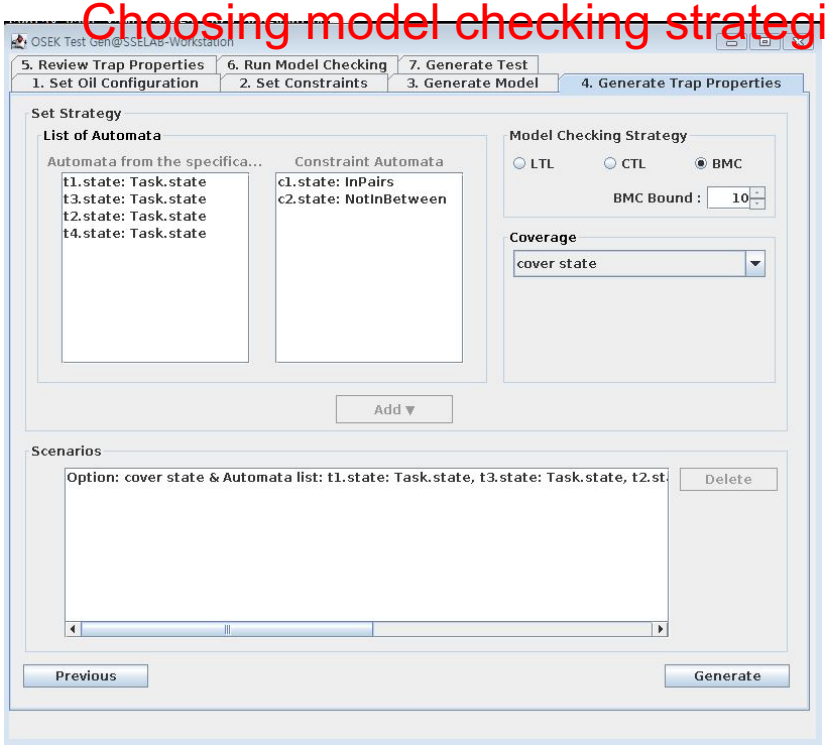
Tool demo available at https://youtu.be/ajhV9N2jCb8

Prototype tools

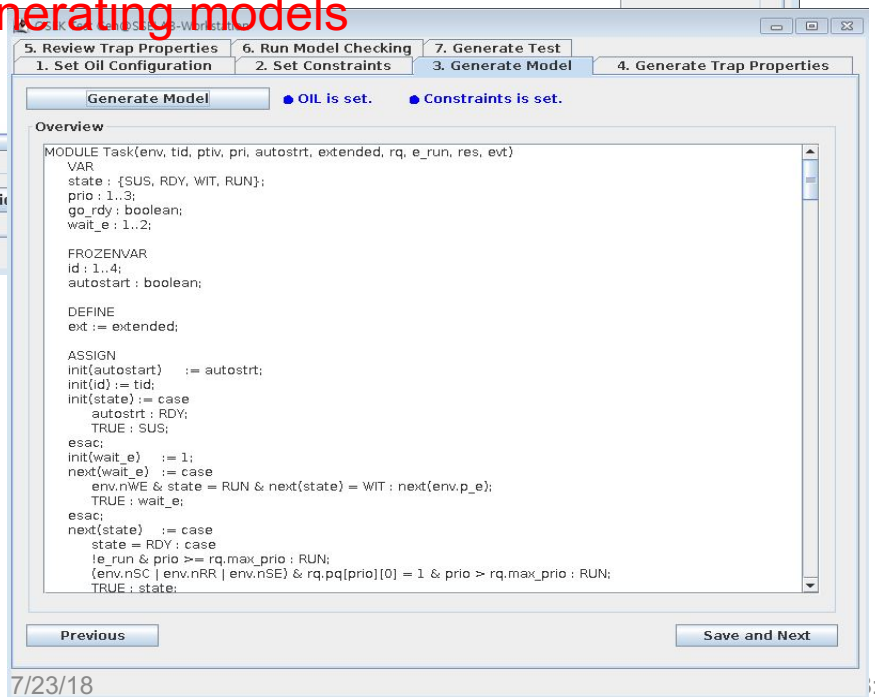
Choosing constraints



Choosing model checking strategies

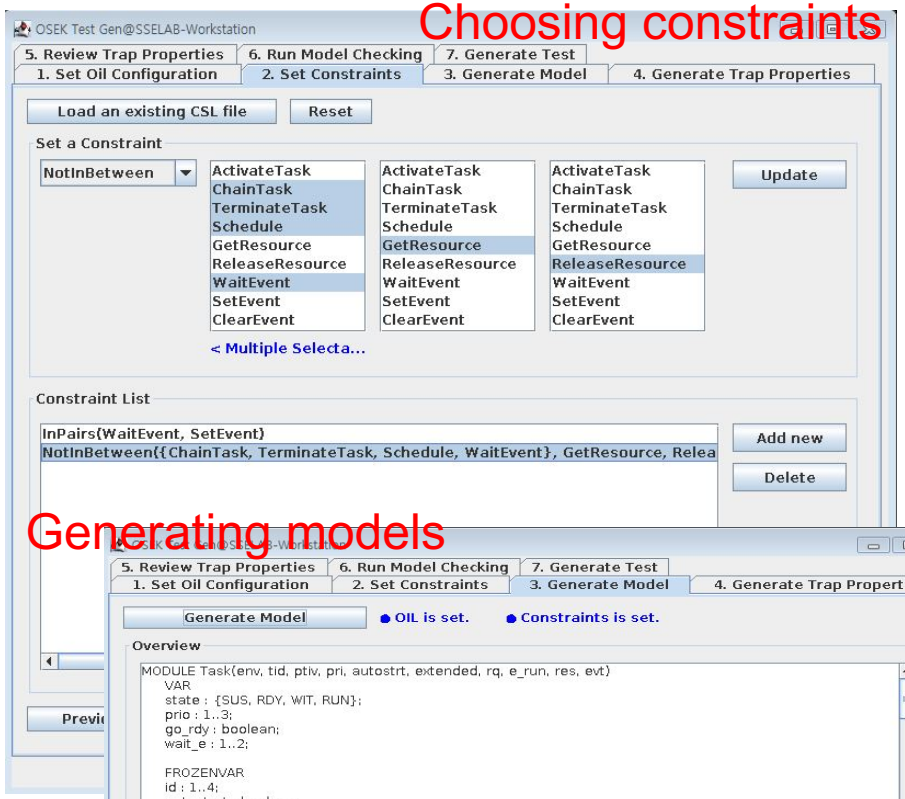


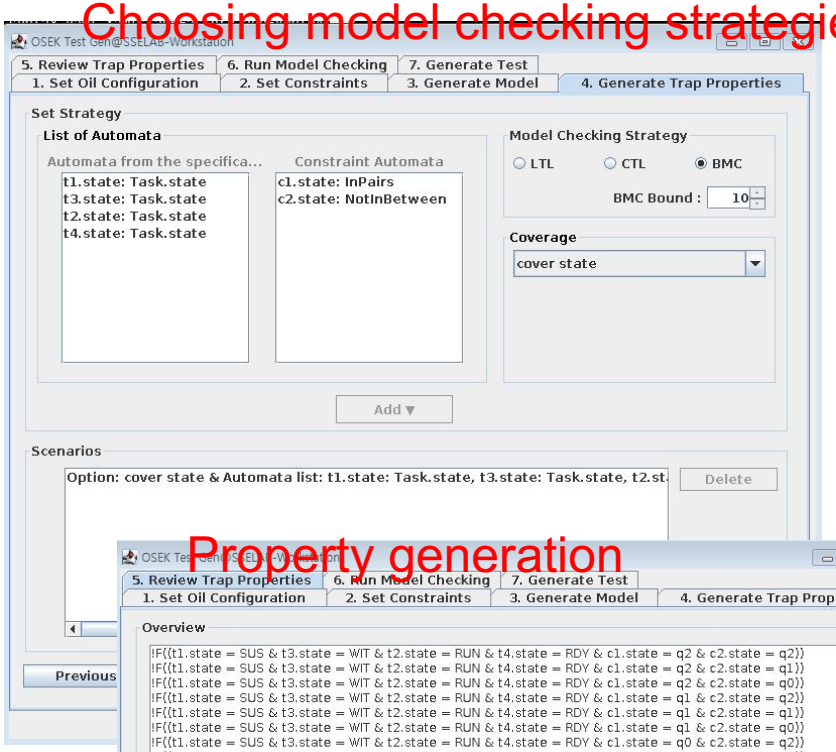
Generating models

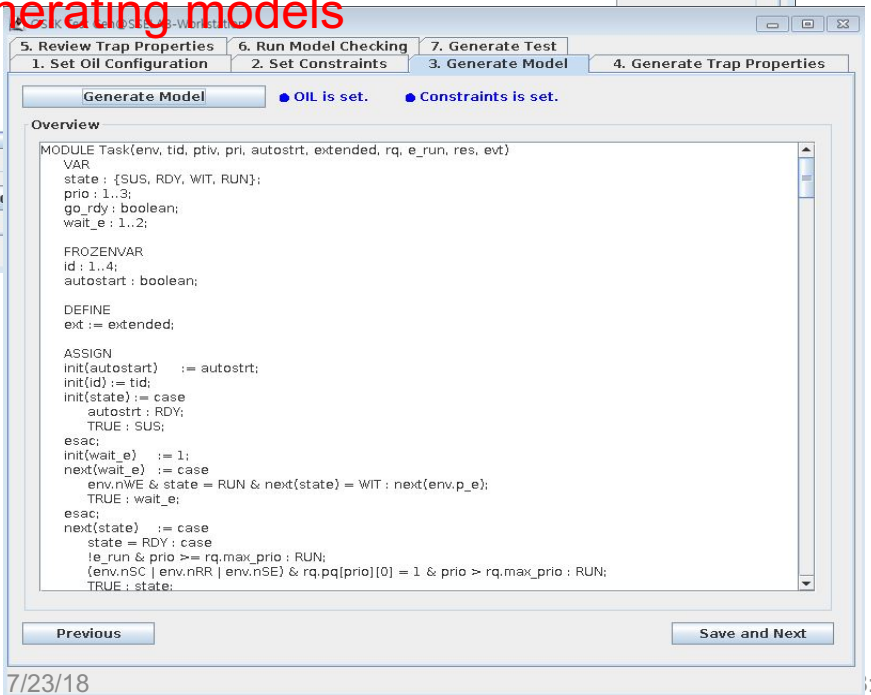


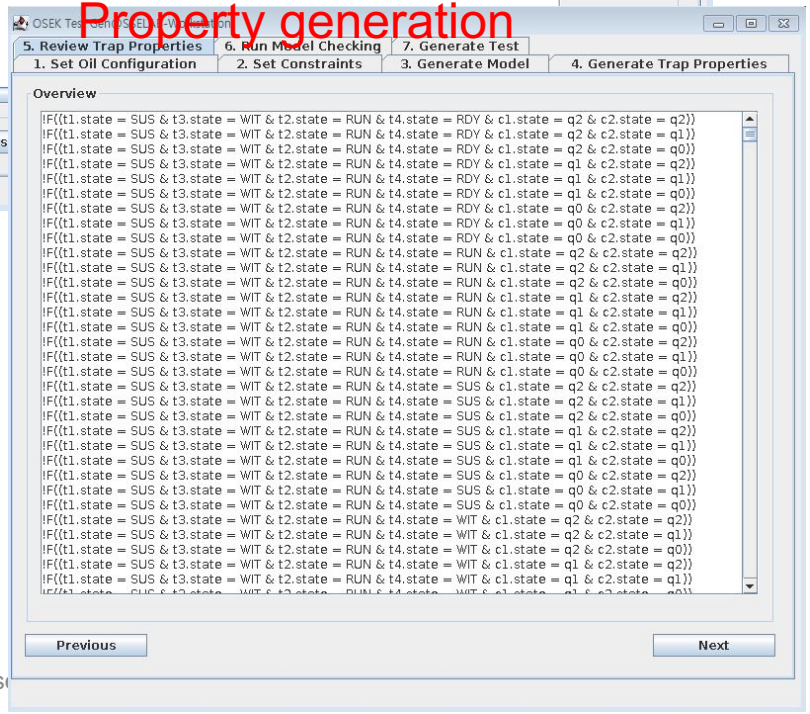
Tool demo available at https://youtu.be/ajhV9N2jCb8

Prototype tools



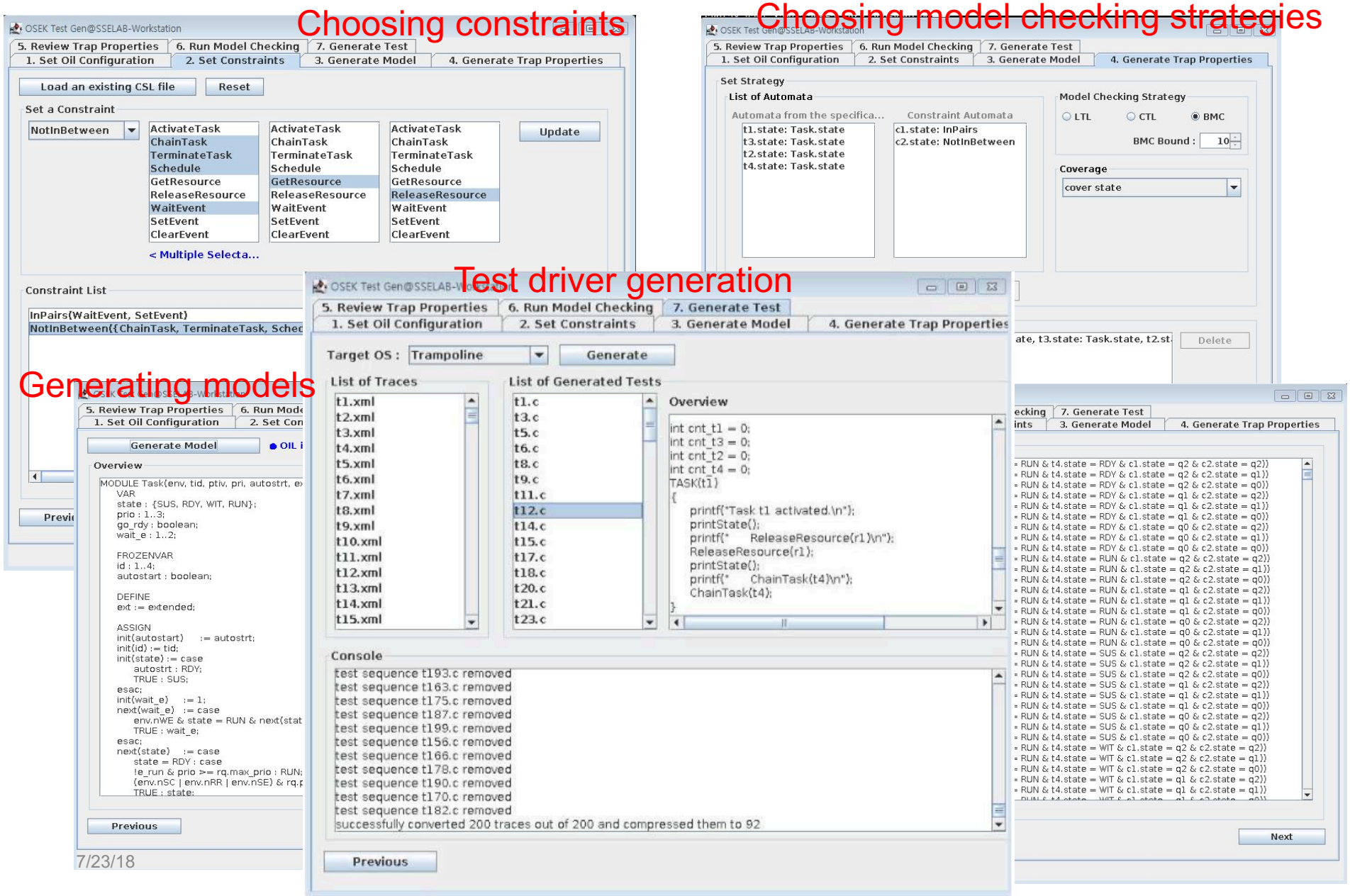






Tool demo available at https://youtu.be/ajhV9N2jCb8

Prototype tools



User's perspective

Configuration

```
OIL_VERSION = "2.5";

CPU default_cpu {
	OS myOS {
		STATUS = EXTENDED;
		APP_SRC = "gen_model_test.c";
		LDFLAGS = "-g -lpthread";
		CFLAGS = "-DWITH_DEBUG";
	};
	APPMODE std { };
	TASK t1 {
		PRIORITY = 1;
		AUTOSTART = TRUE {
			APPMODE = std;
		};
		SCHEDULE = FULL;
		ACTIVATION = 1;
		RESOURCE = r0;
	};
	TASK t2 {
		PRIORITY = 2;
		AUTOSTART = FALSE;
		SCHEDULE = FULL;
		ACTIVATION = 1;
		RESOURCE = r1;
		EVENT = e0;
	};
```

```
int main(void)
{
	StartOS(OSDEFAULTAPPMODE);
	return 0;
}

TASK(t1)
{
	printf("Task t1 activated.\n");
	printState();
	printf("		ActivateTask(t2)\n");
	ActivateTask(t2);
	printState();
	printf("		SetEvent(t1, e0)\n");
	SetEvent(t1, e0);
	printState();
	printf("		TerminateTask()\n");
	TerminateTask();
}


TASK(t2)
{
	printf("Task t2 activated.\n");
	printState();
	printf("		WaitEvent(e0)\n");
	WaitEvent(e0);
}
```

Efficiency of NuSek

- Construction of formal models does not take time
- Test generation from those models is quite efficient
  - No manual effort required
  - Finds more subtle errors than manually constructed test cases by exper ts
    - 64 test cases finding 4 causes of failures vs. 164 test cases finding only one of the 4 causes

Table 3: Performance of test generation

Config.	CS		CS_{all}		CS × TS		$CS \times TS_{all}$	
(T, C)	Time	#P (#S)	Time	#P(#S)	Time	#P(#S)	Time	#P(#S)
(2, 2)	0.073	8 (6)	0.577	18(17)	4.630	64(24)	16.244	128(45)
(2, 3)	0.104	11(7)	147.73	54 (38)	39.461	88 (31)	28.322	176(53)
(2, 4)	0.211	13(7)	209.66	72 (41)	40.480	104(34)	32.152	208(58)
(3, 2)	0.159	11(7)	0.604	18(13)	21.239	132(50)	226.244	704(218)
(3, 3)	0.198	14(10)	23.664	54(33)	56.275	144(59)	370.709	896(253)
(3, 4)	0.186	16(10)	142.475	108(69)	33.629	192(65)	412.688	1024(265)
(4, 2)	0.233	14(10)	1.043	21(15)	64.415	224(92)	6289.913	3584(817)
(4, 3)	0.258	17(12)	93.207	72(42)	103.966	272(109)	3766.217	4352(1102)
(4, 4)	0.271	19(12)	200.365	144(95)	105.308	304(116)	4266.778	4864(1152)

Experiments on Trampoline

- Five constraint patterns are applied including
  1. InPairs(WaitEvent, SetEvent),
  2. InPairs(GetResource, ReleaseResource),
  3. NotInBetween({TerminateTask, ChainTask, WaitEvent, Schedule}, GetResource, ReleaseResource),
  4. MustEndWith(TerminateTask, ChainTask)

No.	Constraint Input	Time	Mem	#TS	#Err	#type	Coverage
1	InPairs(WaitEvent, SetEvent)	2' 22"	248.8MB	4	2	1	22.3%
2	InPairs(WaitEvent, SetEvent) NotInBetween(GetResource, ReleaseResource, ...)	1' 31"	270.0MB	40	20	1	27.7%
3	InPairs(WaitEvent, SetEvent) NotInBetween(GetResource, ReleaseResource, ...) RestrictedCaller(extended, WaitEvent)	51' 05"	623.8MB	129	90	2	34.0%
4	InPairs(WaitEvent, SetEvent) NotInBetween(GetResource, ReleaseResource, ...) RestrictedCaller(extended, WaitEvent) MustEndWith(TerminateTask, ChainTask)	306' 34"	986.2MB	142	69	2	35.0%

Experiments on Trampoline

- More efficient in finding errors compared to typical s pecification-based test generation

No.	Constraint Input	Method	Time	Memory	#Tests	#Failures	Error types	Branch Coverage
1	InPairs(*WaitEvent, SetEvent*) NotInBetween(*{TerminateTask, ChainTask, WaitEvent, Schedule}, GetResource, ReleaseResource*) CallerMode(*extended, WaitEvent*)	CB	7' 39"	881MB	168	101	W,S,R	34.5%
2		SB1	19"	212MB	39	0	_	27.7%
3		SB2	4' 07"	293MB	39	2	S	32.0%
4	InPairs(*WaitEvent, SetEvent*) NotInBetween(*{TerminateTask, ChainTask, WaitEvent, Schedule}, GetResource, ReleaseResource*) CallerMode(*extended, WaitEvent*) CallerMode(*extended, ClearEvent*)	CB	21' 22"	1594MB	444	355	C,W,S,R	35.3%
5		SB1	16"	210MB	39	0	_	29.7%
6		SB2	5' 57"	284MB	39	5	S	34.5%
7	Constraint input for 1~3 + MustEndWith(*ChainTask, TerminateTask*)	CB	68' 15"	4725MB	398	288	W,S,R	33.2%
8	Constraint input for 7 + CallerMode(*extended, ClearEvent*)	CB	187' 42"	7353MB	1094	940	C,W,S,R	35.5%
9	Constraint input for 1~3 + 4 tasks	CB	603' 18"	8355MB	225	69	W,S,R	32.5%

<Comparison with typical specification-based testing approaches>

Identified issues: an example

- Array bound error and null pointer exception (in standard mode)
  - When a task sets an event that does not belong to the task

```
 1.  tpl_task_events t0_task_evts = {0, 0};
 2.  tpl_task_events * const tpl_task_events_table[1] = { &t0_task_evts };
 3.
 4.  tpl_status tpl_set_event(task_id task_id, tpl_event_mask incoming_event){
 5.    …
 6.    tpl_task_events * const events = tpl_task_events_table[task_id];
 7.    if(task->state != (tpl_proc_state)0x0){
 8.      events->evt_set |= incoming_event;
 9.    …
10.  }
```

t1 sets event?

  - OSEK requires only the task owning the event can set the event

Summary

- A **formal specification** can be used
  - ✓ to **define functional patterns** as well as **constraint patterns**
  - ✓ to **generate** smaller number of **test sequences** that **covers both e xpected and unexpected system behaviors**
- A configuration-dependent formal model generation can be aut omated
- Test generation using constraint patterns is efficient and effectiv e
  - ✓ No redundant test sequences
- The use of OS models in test generation reduces the number of infeasible test sequences
  - ✓ No false alarms

References

- Mauro Pezze and Michael Young, *Software Testing and Analysis*, Wiley, 200 8
- Paul Ammann and Jeff Offutt, *Introduction to Software Testing*, Cambridge University Press, 2008
- Donald Firesmith, *Texonomy of Testing*, Technical Report, CMU SEI, 2015
- Olli-Pekka Puolitaival, *Model-based testing tools*, VTT Technical Research C enter of Finland
- Mike Whalen, *Why We Model: Using MBD Effectively in Critical Domains*, I CSE 2013 MiSE tutorial
- Yunja Choi, *A configurable V&V framework using formal behavioral pattern s for OSEK/VDX operating systems*, Journal of Systems and Software, 2018
- Yunja Choi and Taejoon Byun, *Constraint-based test generation for automo tive operating systems*, Software and Systems Modeling, 2017


Q
A