KTH Computer Science
and Communication

Introduction to Model-Based Testing

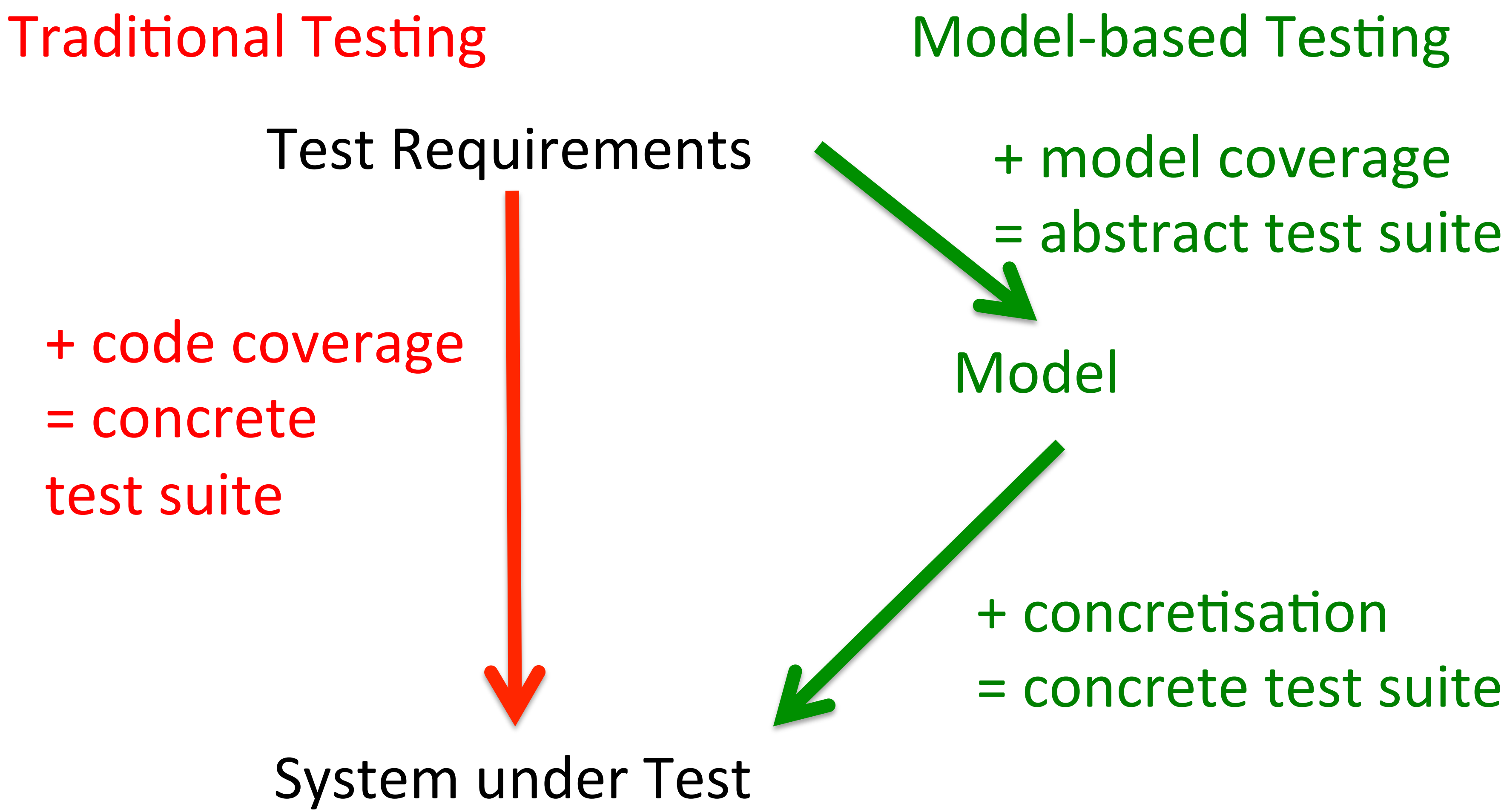
Karl Meinke, CSC School
KTH Stockholm

Overview

- Part 1: basic overview and principles
- Part 2: in-depth study of MBT technologies

Part 1: Basic overview and principles

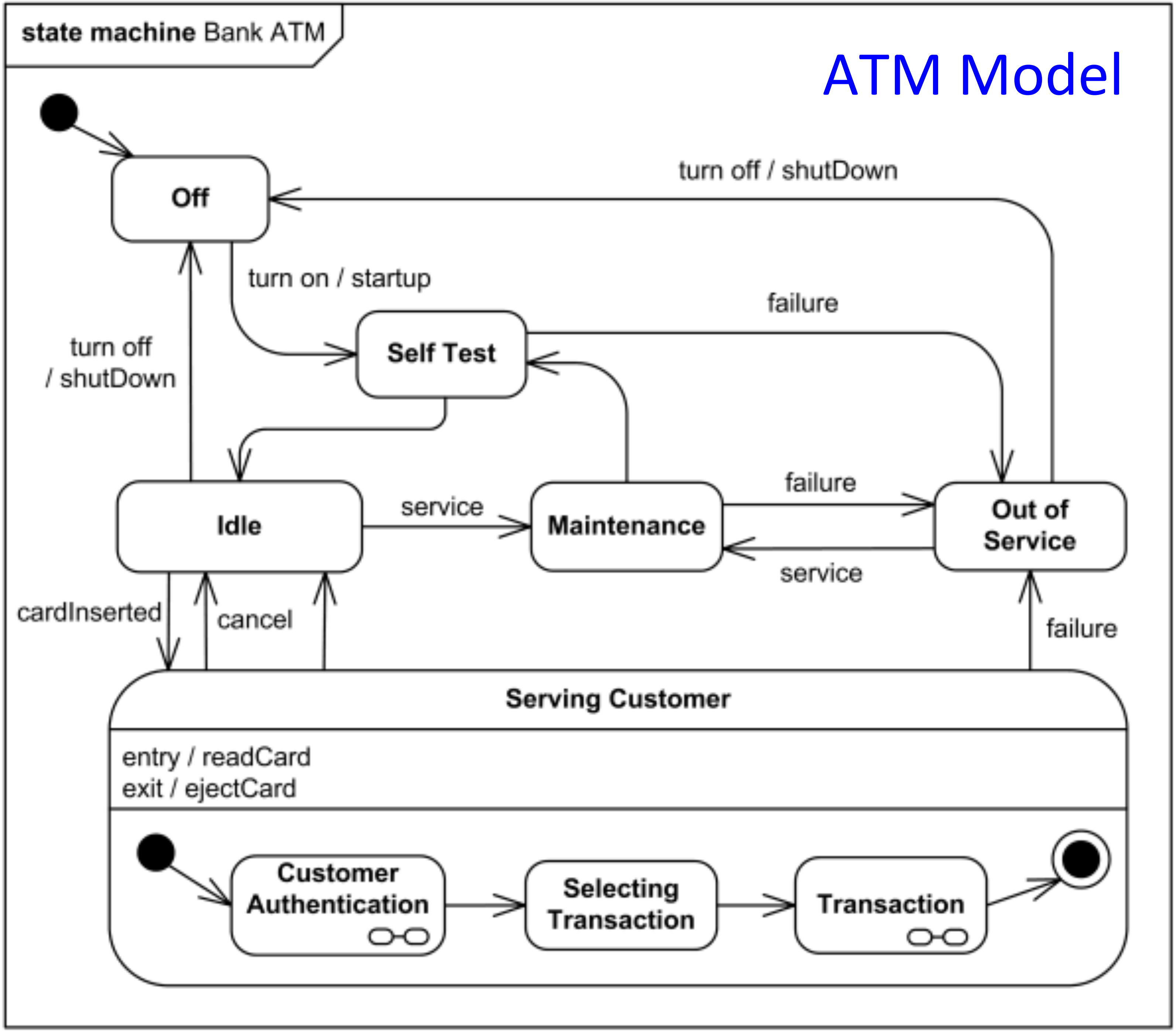
What is Model-based Testing?



What models to use?

- Basically need *dynamic models*
  - Statecharts
  - Sequence diagrams (use cases)
  - Executable code

ATM Model


state machine Bank ATM
Off
turn off / shutDown
turn on / startup
failure
turn off
/ shutDown
Self Test
Idle
service
Maintenance
failure
Out of
Service
service
cardInserted
cancel
failure
Serving Customer
entry / readCard
exit / ejectCard
Customer
Authentication
Selecting
Transaction
Transaction

Solve Coverage Problem?

- Model need only reproduce some features of a system under test
- Simplification of code (abstraction)
- We decide which features! (What to test?)
- False positives and negatives?
- Quicker and easier to generate tests
- Use traditional coverage models (graphs)
- Can be automated
- Tool support (Conformiq, Spec Explorer …)

Solve Oracle Problem?

- Model can be used to determine verdicts
- AKA. conformance testing
- Verdict construction can be reduced to equality or membership test.
- Needs exact synchronisation between model and code
- Difficult with legacy models and agile development.

Conformance Testing (bad news)

- **Claim**: There is a sense in which conformance testing just pushes the testing problem elsewhere
- **Why?**: How do we validate our model?

- Simulation?
- Testing? (systematic simulation?)
- Formal verification? (too big or complex?)

Conformance Testing (practise)

1. Take a system model e.g. statechart
2. Take a coverage model, e.g:
   2.1 Node coverage
   2.2. Edge coverage
3. Construct test cases to reach x% coverage
   3.1. Manually
   3.2. Constraint solver (added value of a tool)
4. Translate test cases into scripts, run, record, and compare

When to use Model-based testing?

- Model-based testing can be conducted as part of model-based development
- Model-based development = describing software using accurate modeling languages e.g. UML

Modeling Maturity Level

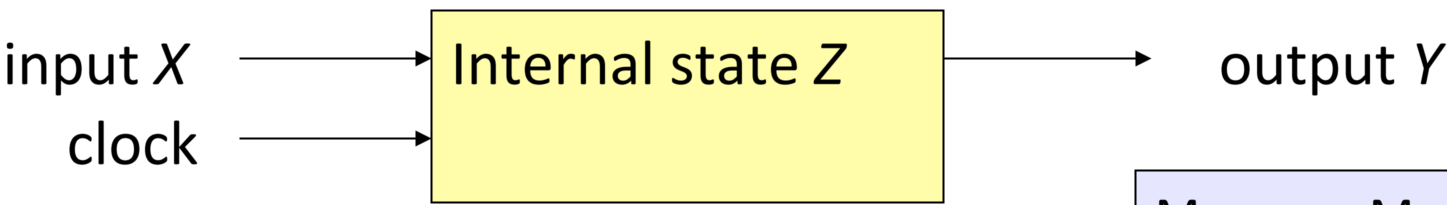
- Level 0: No specification: software specifications are only in the heads of developers
- Level 1: Textual: The software specifications are written down in informal natural language documents
- Level 2: Models with text: a set of models (diagrams or text with well defined meanings). Natural language is used to motivate, explain and detail models. Transition from model to code is manual. Model synching with code is difficult.

- Level 4, Precise models: code can be generated from models, and modified for special requirements. Model has a precise meaning independent of natural language. Natural language still used.
- Level 5, Models only: model is like a high-level programming language. Model-to-code generation automatic (compiler). Generated code used without changes.

UML Statecharts

- A ***UML statechart***, is an object-based variant of Harel's statechart language.
- Statecharts overcome limitations of finite state machines, without losing benefits.
- Combine aspects of Moore and Mealy machines
- New concepts:
  - Hierarchically nested states
  - Orthogonal regions
  - Extended actions
  - History

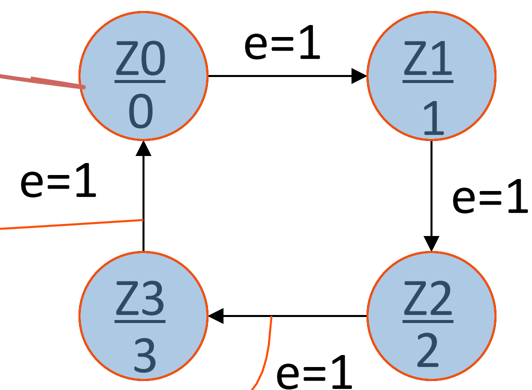
Classical automata



Moore- + Mealy automata=finite state machines (FSMs)

Next state Z^+ computed by function δ
Output computed by function λ

- Moore-automata:
  $Y = \lambda\,(Z); \quad Z^+ = \delta\,(X, Z)$
- Mealy-automata
  $Y = \lambda\,(X, Z); \quad Z^+ = \delta\,(X, Z)$



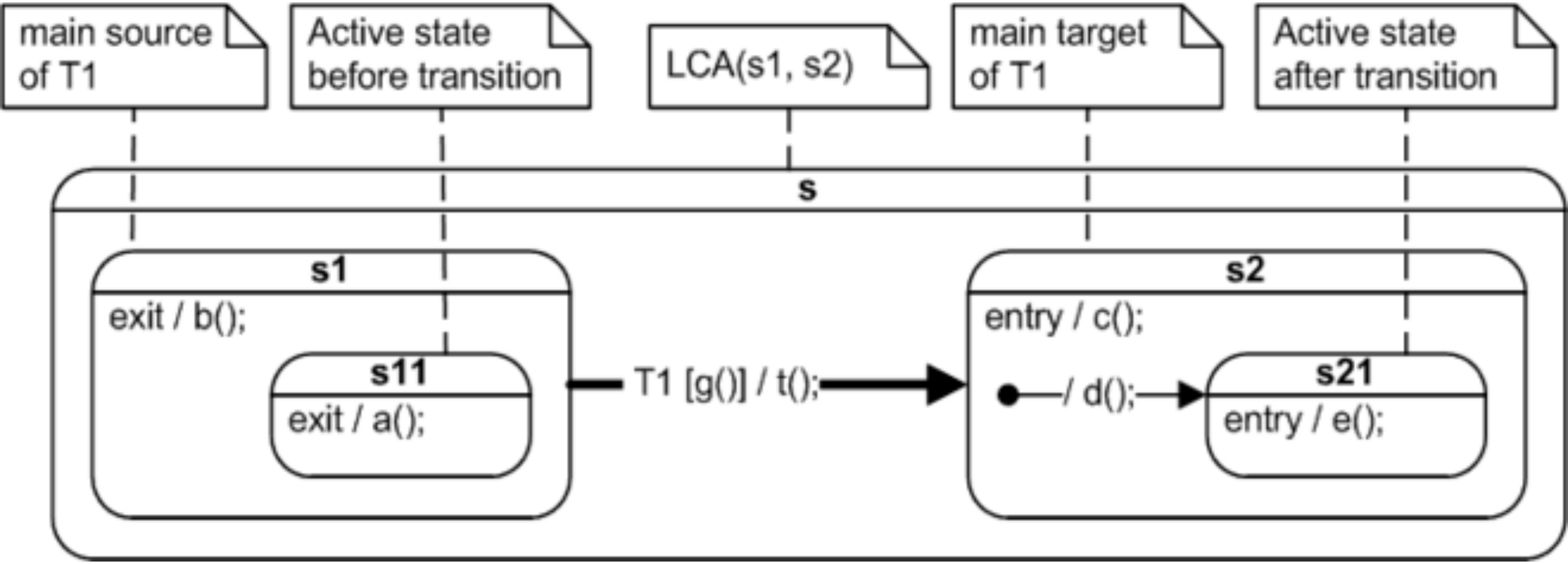
UML 2.4

- Two kinds of state machines.
- Behavioral state machines are used to model the behavior of individual entities (e.g., class instances)
- Protocol state machines are used to express usage protocols and can be used to specify the legal usage scenarios of classifiers, interfaces, and ports.
- Behavioral state machine is subclassed by protocol state machine.

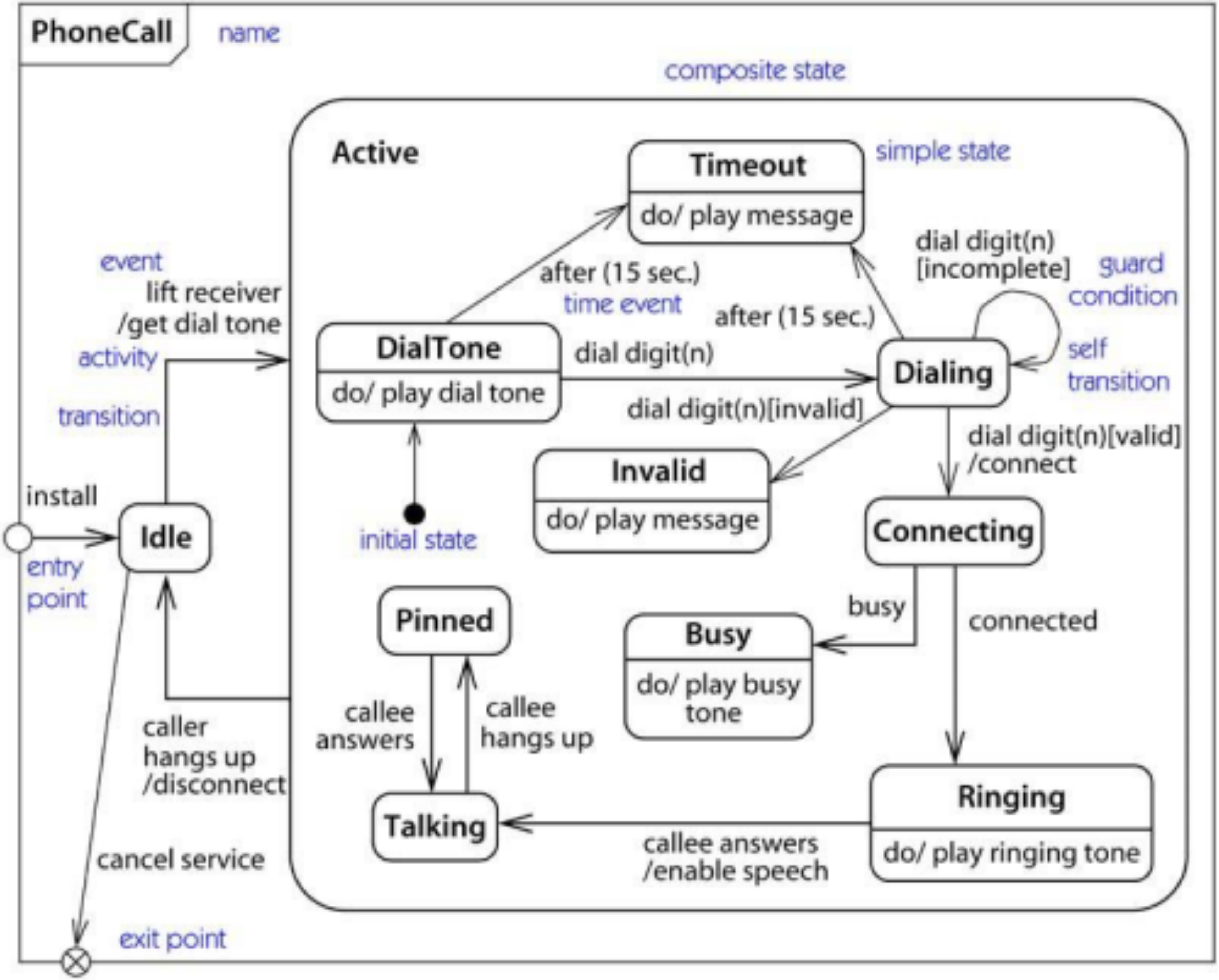
Execution order

- UML specifies that taking a state transition executes the following actions in the following sequence

1. Evaluate the guard condition associated with the transition and perform the following steps only if the guard evaluates to TRUE.
2. Exit the source state configuration.
3. Execute the actions associated with the transition.
4. Enter the target state configuration.



Taking T1 causes the evaluation of guard g();
Exit of s11, s1,
Action sequence a(); b(); t(); c(); d(); and e();
Entry of s2, s21
(assuming guard g() evaluates to TRUE)


PhoneCall
name
composite state
Active
Timeout
do/ play message
simple state
dial digit(n)
[incomplete]
guard
condition
event
lift receiver
/get dial tone
after (15 sec.)
time event
after (15 sec.)
activity
DialTone
do/ play dial tone
dial digit(n)
Dialing
self
transition
transition
dial digit(n)[invalid]
dial digit(n)[valid]
/connect
Invalid
do/ play message
install
Idle
initial state
Connecting
entry
point
busy
connected
Pinned
Busy
do/ play busy
tone
callee
answers
callee
hangs up
caller
hangs up
/disconnect
Talking
Ringing
do/ play ringing tone
callee answers
/enable speech
cancel service
exit point

MS Spec Explorer (Spec#)

```
Class Client {
    bool entered;
    Map<Client,Seq<string>> unreceivedMsgs;

    [Action] void Enter()
    Action of abstract state machine
        requires !entered; {  required condition
            entered = true;
    }
```

```
[Action] void Send(string message)
    requires entered; {
    foreach (Client c in enumof(Client), c != this,
    c.entered)
        c.unreceivedMsgs[this] += Seq{message};
}
```

A Chat room model

Part 2: In-depth study of MBT technologies

Model-based Automated Test-Case Generation (ATCG)

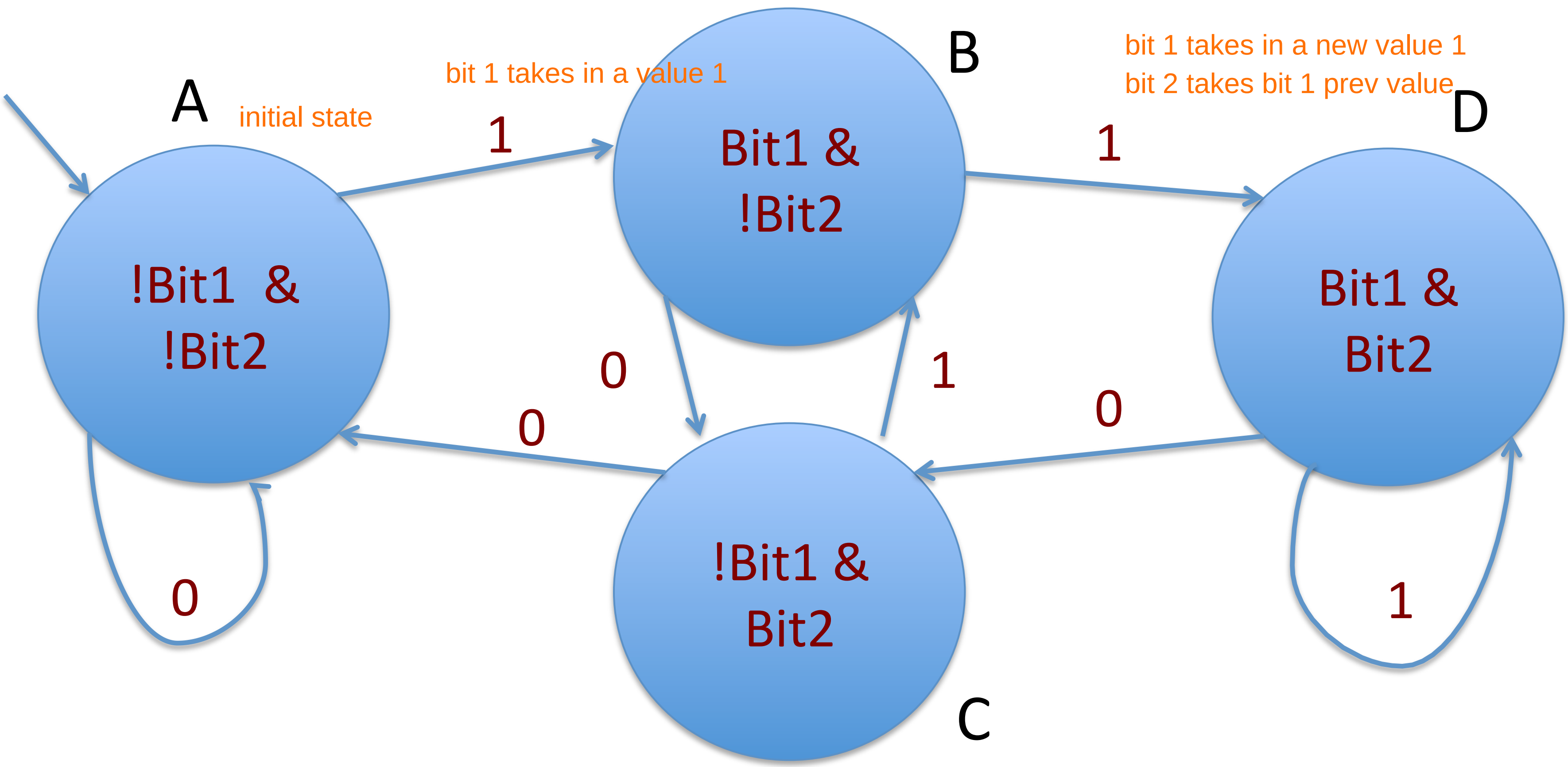
- Manual construction of test cases is a difficult, time-consuming and error-prone activity that requires expertise.
- Automated Test-Case Generation (TCG) has been the "holy grail" of testing for some time.
- Is this even possible? If so, how?
- Black/white-box testing
- Want algorithms which generate test cases (preferably with a known level of coverage)

Model Checking

- A model checker is a tool which takes as input
  - An automaton model M
  - A logical formula ϕ
- If ϕ is a true statement about all possible behaviors of M then the model checker confirms it (proof)
- If ϕ is a false statement about M the model checker constructs a counterexample (a simulation sequence)
- A counterexample to ϕ satisfies !ϕ
- A simulation sequence can be executed as a test case

Two Bit Shift Register

- $Q = \{A, B, C, D\}$, $\Sigma = \{0,1\}$, $q_0=A$



2-Bit Shift Reg in .smv format

symbolic model verifier

```
MODULE main
VAR
-- system outputs
   Bit1 : boolean; -- Boolean variable
   Bit2 : boolean;
   state : {A, B, C, D}; -- scalar variable

IVAR
-- system inputs
    input : boolean;

ASSIGN
   init(state) := A;
   init(Bit1) := 0;
   init(Bit2) := 0;
```

initialisation of initial value

```
next(state) := case
   state = A & input = 1 : B;
   state = B & input = 0 : C;
   state = B & input = 1 : D;
   state = C & input = 0 : A;
   state = C & input = 1 : B;
   state = D & input = 0 : C;
   TRUE : state;
esac;
   next(Bit1) := case lab 3 esac;
   next(Bit2) := case lab 3 esac;
```

if in state A and takes an input 1, will go to state B

every line here correspond to an arrow in the model

capture self transition

Linear Temporal Logic LTL in .smv syntax

- Use temporal logic to express automaton requirements

- Boolean variables does not give sense of time
- `A, B, …, X, Y, .. MyVar`, etc.
- Boolean operators does not give sense of time
- `!(`ϕ`), (`ϕ` & `θ`), (`ϕ` |`θ`), (`ϕ` -> `θ`)` ,…
- Temporal (time) operators LTL Operators
- `F(`ϕ`)` (sometime in the future ϕ) F doesn't say how long the gap is, how far into the future is
- `G(`ϕ`)` (always in the future ϕ)
- `(`ϕ` U `θ`)` (ϕ holds until θ holds)
- `X(`ϕ`)` (next ϕ holds) we do not know how long the sequencing in 'next'
- Write $X^n(\phi)$ for $X(X(\ldots X(\phi)))$ (ϕ holds in n steps)

F,G,X are always forward looking -- looking into the future.
They didn't capture the past.

Useful Logical Identities for LTL

- Boolean identities

  `!(!(φ)) ⇔ φ,     !(ϕ | φ) ⇔(!ϕ & !φ)` ,

  `(ϕ -> φ) ⇔(!(ϕ) | φ)` etc.

- LTL identities

  `!(G(!(ϕ))) ⇔ F(ϕ)`

  `!(X(ϕ)) ⇔ X(!(ϕ))`

  `G(φ & ϕ) ⇔ G(φ)& G(ϕ)`

  `G(φ) ⇔ φ & X (G(φ))`

  `G (G(φ)) ⇔ G(φ)`

- <u>Exercise</u>: using these identities, prove:

  `!(F(!(φ))) ⇔ G(φ)`

  `F(φ | ϕ) ⇔ F(φ) | F(ϕ)`

  `F(φ) ⇔ φ | X(F(φ))`

- Remark TCG usually involves negating formulas, so its useful to <u>understand what a negation means</u>

Examples

Right now it is Wednesday

Wednesday

Tomorrow is Wednesday

X (Wednesday)

(A) Thursday (always) immediately follows Wednesday

G(Wednesday -> X (Thursday))

(A) Saturday (always) follows Wednesday

G(Wednesday -> F(Saturday))

Yesterday was Wednesday

it is always that if and only if tomorrow is thursday then today is wednesday; and that if today us wednesday, tomorrow is thursday; today is thursday

G(Wednesday <-> X (Thursday)) & Thursday

- Exercise: define the sequence of days precisely, i.e. just one solution
- Question: are there any English statements you can't make in LTL?
- Question: what use cases can you express in LTL?

LTL Specifications in .smv files

```
LTLSPEC
  G( Bit1 <-> (X Bit2) )
  -- always the value of Bit1 now equals the
  -- value of Bit2 in the next time step
  -- This is obviously TRUE!

  G( Bit1 <-> (X Bit1) )
  -- always the value of Bit1 now equals the
  -- value of Bit1 in the next time step
  -- This is obviously FALSE!
```

doing something with temporal logic

because of pipeline effect. Bit 1 is feeding Bit 2

NuSMV Output Example

```
-- specification G( Bit1 <-> (X Bit2) )
-- is true
-- specification G( Bit1 <-> (X Bit1) )
-- is false
-- as demonstrated by the following execution
sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
state = A Bit1 = false Bit2 = false
-> Input: 1.2 <-
input = 1
```
constraint solver input, never initialised
```
-> State 1.2 <-
state = B Bit1 = true Bit2 = false
```

Automated White-box TCG

- We can use a model checker to generate counterexamples to formulas (i.e. test cases) with specific structural properties.
- This is done by inspecting the graph structure of the automaton
- i.e. white/glass box testing
- "Most" use of model checkers concerns this.

Test Requirements/ Trap Properties

- Recall a test requirement is a *requirement that can be satisfied by one or more test cases*
- Basic idea is to capture each test requirement as an LTL formula known as a "trap property"

Example Suppose the test requirement is "*cover state D of shift register*" D ≡ Bit 1 & Bit 2

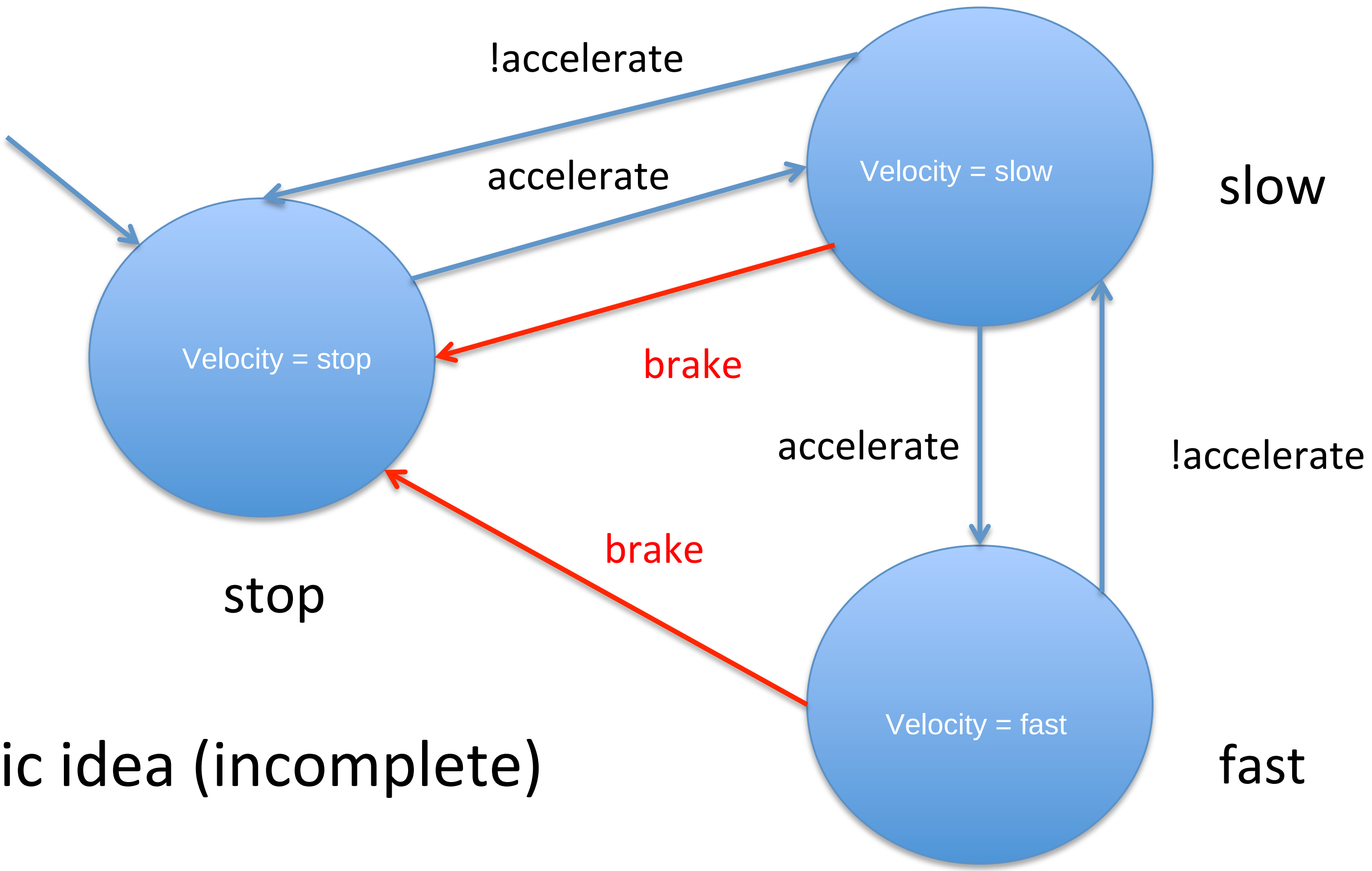
G (! (Bit 1 & Bit 2))

Trap property is `G(!( state = D ))`

This formula is `False` and any counterexample must be a path that goes through state `D`.

If we got a test requirement that covers D, we can use this tool to get a path that steer ourselves to D.

Case Study: Car Controller Model (CC)



Basic idea (incomplete)

```
MODULE main
VAR
   state: {stop, slow, fast}; -- velocity states

IVAR
   accelerate: boolean; -- gas pedal
   brake: boolean; -- brake pedal

ASSIGN
   Init(state) := stop;

   Next(state) := case
      accelerate & !brake & state = stop : slow;
      accelerate & !brake & state = slow : fast;
      !accelerate & !brake & state = fast : slow;
      !accelerate & !brake & state = slow: stop;
      brake: stop;
      TRUE: state;
   esac;
```

Trap properties for Structural Coverage Models

- Let's use NuSMV to automatically construct test suites that satisfy the different structural coverage models introduced in Lecture 2.
- Examples:
  - Node coverage NC
  - Edge coverage EC
  - Condition coverage PC
- How to interpret these concepts?

Node Coverage for CC

- Want a path that visits each node
- Simple approach: write 1 trap property per node
- General form:
- `G(!(state = <state_name>))` G(! ("unique_state_property"))
- Counterexamples satisfy:
- `F(state = <state_name> )` F("unique_state_property")
- Example:

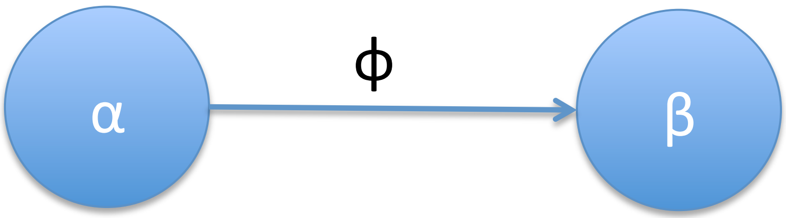
```
G(!(state = stop));
```

G(!(velocity = stop));
G(!(velocity = slow));
G(!(velocity = fast));

- Clearly this will give redundant test cases, but method is still linear in state-space size.
- Lab Exercise 3: define the remaining 2 trap formulas for car controller

Edge coverage for CC

- Want to traverse each edge between any pair of nodes



- General form G((state = α & ϕ) -> X (!(state = β))
- Counterexample satisfies F(state = α & ϕ & X(state = β))
- Example:

```
G(state=stop & accelerate ->
      X (!(state=slow)) )
```

G(velocity=stop & accelerate -> X(!(velocity =slow)))

- Lab Exercise 3: define the remaining 5 trap formulas for car controller

Requirements-based TCG

- Can we also perform requirements-based test case generation?
- Want to test the requirements are fulfilled rather than explore structure of the code.
- Can look at negation of a requirement

Car Controller LTL requirements

1. Whenever the brake is activated, car has to stop quickly

```
G(brake -> X(state=stop) )
```

2. When accelerating and not braking, velocity has to increase gradually

```
G(!brake & accelerate & state=stop ->
           X(state=slow) )
G(!brake & accelerate & state=slow ->
           X(state=fast) )
```

3. When not accelerating and not braking, velocity has to decrease gradually

```
G(!brake& !accelerate & state=fast ->
           X(state=slow) )
G(!brake & accelerate & state=slow ->
           X(state=stop) )
```

Safety Requirements

- A safety requirement describes a behavior that may not occur on any path.
- "*Something bad never happens*"
- To verify, all execution paths must be checked exhaustively
- Safety properties usually have the form `G!` ϕ where ϕ defines the "*bad thing*"
- Counterexamples (test cases) are finite

Liveness Requirements

- A liveness requirement describes a behavior that must hold on all execution paths
- "Something good eventually happens"
- Safety does not imply liveness
- Liveness does not imply safety
- Liveness properties often have the form

  $F(\theta)$ or $G(\phi \rightarrow X^n\theta)$ or $G(\phi \rightarrow F\theta)$

  where θ describes the "good" thing and ϕ is some precondition needed for it to occur.
- Counterexamples may be finite or infinite (why?)

TCG for Model-Based Requirements Testing

- Suppose we have an LTL requirement ϕ
- Feed into NuSMV an FSM model A of the SUT, together with the negated formula !ϕ

Finite State Machine

- Choose any counterexample (a behaviour b) (there should be lots if A is a correct model and is a correct requirement).
- b satisfies !! ϕ i.e. b is an example of correct behavior
- Feed the inputs of b into the SUT and observe the output
- If output from the SUT matches b then pass else fail

Car Controller Examples

(1) `F(brake & X !(velocity=stop))`

(2) `F(!brake & accelerate & velocity=stop -> X(velocity=slow))`

Conclusions

- Model-based testing has created a lot of interest and opened up new questions
- Tools market is emerging
- Conceptual problems about models
- Automated TCG is possible using off-the-shelf model checkers
- New methods such as learning-based testing avoid manual model construction.