Formal Conformance Testings 2006

Lecture 10

2nd November 2004

Specification-based testing algorithms

- Algorithms for running testing, based on a specification

Basic on-the-fly algorithm

$E := \emptyset$, $C := 0$
repeat
    $X := \{ \langle E \cup \langle m, C\rangle, C+\epsilon\rangle \mid m \in \Sigma, \epsilon > 0, \langle E \cup \langle m, C\rangle, C+\epsilon\rangle \in Tr(S) \}$
wait:
    $X_T := \{ \langle E, C+\epsilon\rangle \mid \epsilon > 0, \langle E, C+\epsilon \rangle \in Tr(S) \}$
    $N := X_T \cup X$
    if [$N = \emptyset$] then FAIL
    if [stopping criterion] then PASS
    choose $T = \langle E', t\rangle$ from N
    if $T|_C \in \Sigma_{in}$ then { send $T|_C$, $E := E \cup \langle T|_C, C\rangle$ }
    wait for input until t // note: $t > C$
    if [input m received at time t' ($C \leq t' < t$)]
        then $E := E \cup \langle m, t'\rangle$; $C := t'$; $X := \emptyset$; goto wait
        else $C := t$

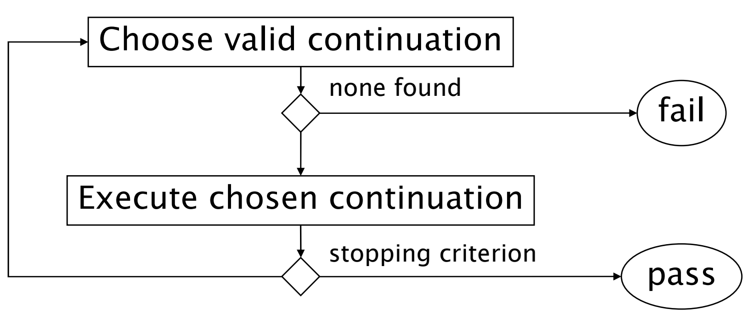
Correctness arguments

- $\langle E, C\rangle$ is "current" trace
- If there is no proper extension of $\langle E, C\rangle$ in Tr(S), we give FAIL verdict
    - FAIL or ERROR is correct, must show that ERROR is unnecessary
- Otherwise we "hypothesize" an extension of at most one, immediately occurring extra event
    - If the event is input to SUT, we produce that
    - The extension is legal (in Tr(S))
- We wait until the end of the extension
- If SUT produces events, these are recorded
- We now claim that ERROR verdict cannot result

Errors?

- At beginning of iteration i, there is at least one extension until some time C+ϵ, otherwise FAIL is signalled
- Suppose on <u>next</u> iteration i+1 the algorithm find empty N, i.e. observed trace is outside specification
- Because the extension chosen by the algorithm is always valid by construction, an output event must have occurred or missed
- This happens always after the possible input event has been sent
- Therefore all deviations from Tr(S) are attributable to output errors

Abstract version



Choosing test steps

- How to choose a test step = how to choose next continuation = testing heuristic
- Where to focus
- Where to "lead" the system under test

Overview

- This is a planning problem
- Assume we can somehow attach "value" to executed test runs
- Test runs that exercise "important parts" of the specification have more value
- We want to create a plan of correct test execution that results in a test run with high value
- But note that we don't know what the SUT will do!

Planning types

- Conformant planning = linear plan that achieves its goal, <u>no matter what the SUT does</u>
- Single-agent planning = co-operative planning = plan that assumes that SUT <u>co-operates</u>
- Adversarial planning = planning against enemy = plan that assumes that SUT actively <u>resists testing</u>
- Stochastic planning = planning against nature = plan that assumes that SUT makes its own choices stochastically

Example

- Test that you can get 6 by throwing die
- <u>Conformant plan</u>: none, as there is no way to enforce the die to give 6
- <u>Single-agent plan</u>: roll once—the die will co-operate and give 6
- <u>Adversarial plan</u>: no plan—how many times you roll, the die will always give something else than 6
- Stochastic plan: roll the die until you get 6—the expected number of rolls is 6

Computational aspects

- Planning in general is very difficult
- Conformant plans do not always exist
- Single-agent planning is in practice cheaper than adversarial or stochastic planning

Discussion

- In practice SUTs are not co-operating nor adversarial; they are independent and stochastic, but their stochastic choice functions are not known
- Co-operative planning is a "quick heuristic"
- Adversarial planning is "worst case analysis" which guarantees in theory best worst-case performance—but is computationally very expensive
- Conformant planning only for simple systems

When to stop testing?

- Two heuristic problems in testing
  - What to do
  - When to stop
- If you have arbitrarily much time, you should test arbitrarily long
- In practice there is a trade-off between better testing and spending more resources
- This is the “stopping criterion”
- Trade-offs can be analyzed using rational decision theory and like theories
  - More on this later

A goal-oriented version

- A test execution algorithm that “aims” at a specific trace
- The trace is chosen by the algorithm, in a yet unspecified manner

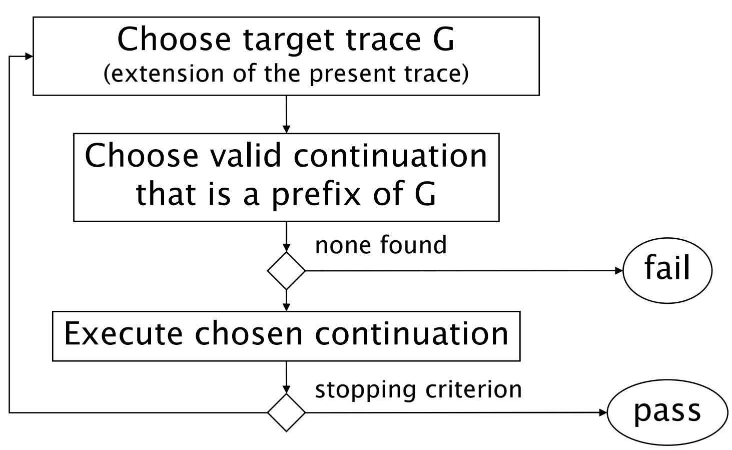
Copyright © Antti Huima 2004-06. All Rights Reserved.

Basic on-the-fly algorithm

E := ∅, C := 0
repeat
    X := { <E ∪ <m, C>, C+ε> | m ∈ Σ, ε > 0, <E ∪ <m, C>, C+ε> ∈ Tr(S) }
wait:
    X_T := { <E, C+ε> | ε > 0, <E, C+ε> ∈ Tr(S) }
    Choose a suitable G from Tr(S) s.t. <E, C> is proper prefix of* G
    N := (X_T ∪ X) ∩ pfx(G)
    if [N = ∅] then FAIL
    if [stopping criterion] then PASS
    choose T = <E', t> from N
    if $T|_C \in \Sigma_{in}$ then { send $T|_C$, E := E ∪ < $T|_C$, C> }
    wait for input until t   // note: t > C
    if [input m received at time t' (C ≤ t' < t)]
        then E := E ∪ <m, t'>; C := t'; X := ∅; goto wait
        else C := t

* because E may contain event e at time C, in this case we must check that <E–e,C> is prefix of G and that G contains e

Abstract version



Comments

- Decision about “where to proceed” has been factored into two decisions:
  - What is the aim
  - What is the next step towards the aim

Property covering

- Assume there exists a universe of “properties”, and a procedure Universal_Property_Check that maps a trace and a specification to a set of properties
  - A set of properties that every “execution” of a specification (as a reference implementation) that produces the given trace has

Property covering (ctd.)

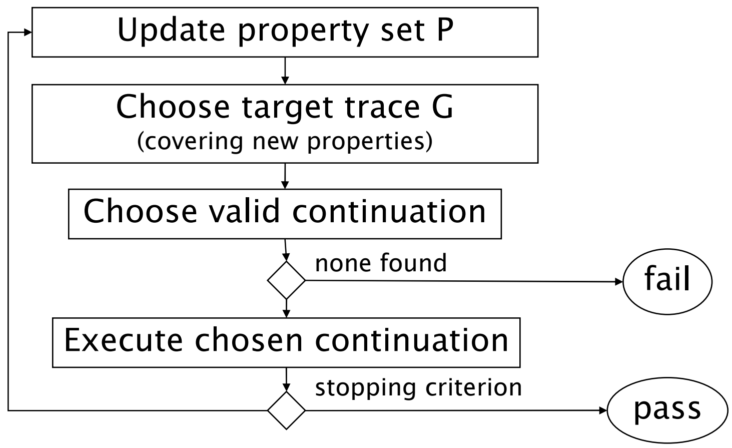
- Furthermore, assume there exists another procedure Plan_For_More_Properties that maps a set of properties, a trace, and a specification, to a new “goal” trace, such that an execution leading to the trace covers more properties
- We get a greedy property-covering testing algorithm

Basic on-the-fly algorithm

```
E := ∅, C := 0; P := ∅
repeat
     X := { <E ∪ <m, C>, C+ε> | m ∈ Σ, ε > 0, <E ∪ <m, C>, C+ε> ∈ Tr(S) }
wait:
     X_T := { <E, C+ε> | ε > 0, <E, C+ε> ∈ Tr(S) }
     P := P ∪ Universal_Property_Check(<E, C>, S)
     G := Plan_For_More_Properties(P, <E, C>, S)
     if [ no G found ]
          Choose a suitable G from Tr(S) s.t. <E, C> is proper prefix of* G
     N := (X_T ∪ X) ∩ G
     if [ N = ∅ ] then FAIL
     if [ stopping criterion ] then PASS
     choose T = <E’, t> from N
     if T|_C ∈ Σ_in then { send T|_C, E := E ∪ < T|_C, C> }
     wait for input until t                    // note: t > C
     if [ input m received at time t’ (C ≤ t’ < t) ]
          then E := E ∪ <m, t’>; C := t’; X := ∅; goto wait
          else C := t
```

* because E may contain event e at time C, in this case we must check that <E–e,C> is prefix of G and that G contains e

Abstract version



Summary

- Basic on-the-fly algorithm
- Planning types
- Stopping criterion
- Goal-oriented testing

Formal Conformance Testing 2006

Lecture 13

9th Nov 2006

Interpreting programs as specifications

- A program (e.g. Java + UML program) is interpreted as a specification by considering it as a *reference implementation*
- Any behaviour that the reference implementation can produce is valid
- Any behaviour that the reference implementation could not produce is invalid

Notation

- Denote by ETr(p) the set of execution traces the program can generate
- ETr(p) assumed prefix-complete by construction
- Denote by Tr(p) the largest subset of ETr(p) that is serial

Computational view

- Given a program p and a trace T, it is difficult to check if $T \in Tr(p)$, from a computation point of view
  - Checking $T \in ETr(p)$ is an unsolvable problem (→ infinite state model checking)
  - Checking $T \in Tr(p)$ *additionally* requires checking that there exists at least one family of arbitrarily long extensions of T

Computational view continued

- Using Tr(p) as a set of valid traces causes thus some real world complications—in the general case
- But if program p e.g.
  - always accepts all inputs, and
  - never crashes,
- then Tr(p) = ETr(p), and we are left “only” with the trace inclusion check

A dive deeper

- How do we check if T $\in$ Tr(p) for a given program p?
- How do we compute the ”properties” that a trace ”necessarily” covers?
- How do we compute goal traces?

State space based computation

- Tr(p) (for a program p) is <u>external behaviour</u>. It abstracts away the "internals" of the program
- This is not practical from the computation point of view
- Typically also the internal and "silent" computation steps count and cause difficulties
- → internal state spaces

State spaces

- A state is (here) a pair <c, T> where c is an "internal control state" and T is an I/O trace produced "until now"
- For every state s, there exists a set of successor states (potentially infinite), denoted by next(s)
- If s' ∈ next(s), we write also s→s'

State spaces

- Assume we can associate with a specification program
  - an initial state $s_0=\langle c_0,\langle\emptyset,0\rangle\rangle$
  - next state relation
- $ETr(p) = \{ T \mid \exists\langle c,T\rangle : s_0 \rightarrow^* \langle c,T\rangle \}$
- $Tr(p)$ = maximal serial subset of $ETr(p)$
  - In practice we can sometimes assume that the seriality requirement is fulfilled implicitly i.e. $ETr(p) = Tr(p)$

Basic trace inclusion check algorithm

```
W := {s0}
V := ∅
While W≠∅
   Choose <c,T> from W
   If T = T*
        Return FOUND
   Else if T ≺ T*
        V := V ∪ {<c,T>}
        W := W ∪ (next(<c,T>) – V)
   W := W – {<c,T>}
Return NOT FOUND
```

Comments

- If next(s) is infinite, won’t work
  - Symbolic methods needed
- Does not necessarily terminate if
  - Infinite branches (next(s) infinite)
  - Arbitrarily many computation steps possible in finite real time (unboundedly many steps possible before trace end time stamp reaches a constant t)

Properties

- Suppose we can attach a set of properties P to every transition from s to s’
- Write $s \rightarrow_P s'$ if there is a transition from s to s’ with properties P

Universal_Property_Check(T*,S)

```
W := {<s0, ∅>}
V := ∅
P := everything
While W≠∅
    Choose <<c,T>,π> from W
    If T = T*
        P := P ∩ π
    Else if T ≺ T*
        V := V ∪ {<<c,T>, π>}
        N := { <s', π'> | s →Q s', π' = π ∪ Q }
        W := W ∪ (N – V)
    W := W – {<<c,T>,π>}
If P is everything
    Return Trace not found
Else
    Return P
```


Comments

- Computes the set of properties that every execution that produces a given trace must have

Plan_For_More_Properties(P,T*,S)

```
W := {<s0, ∅>}
V := ∅
While W≠∅
    Choose <<c,T>,π> from W
    W := W - {<<c,T>,π>}
    If T ≼ T* or T* ≼ T
        If π ⊈ P and T* ≺ T
            If (Universal_Property_Check(T,S) ⊈ P)
                Return T
        Else
            V := V ∪ {<<c,T>, π>}
            N := { <s', π'> | s →Q s', π' = π ∪ Q }
            W := W ∪ (N - V)
Return Trace not found
```


Comments

- Finds a trace that implies properties that are not present in the set P
- Before the Universal_Property_Check, it holds that at least one way to reach the trace T implies new properties
- The Universal_Property_Check call is used to ensure that this holds for all alternative executions as well

Discussion

- Property = interesting feature in specification
- For example, a property = a state in a state chart model, or a method call in a Java class
- Intuition: it is good to exercise “many parts” of reference implementation rather than “few parts”
- But…

Discussion (ctd)

- … in general it is impossible to <u>prove</u> that this is a good idea
- So just a heuristic

Properties = coverage measures

- Known or used ways to measure “coverage” (properties)
  - Transitions of a state chart
  - States of a state chart
  - Lines visited
  - Branch coverage (true and false branches of switches)
  - Condition coverage (true and false valuations of “atomic” subexpressions in switch expressions)
  - …

Improvements

- Greedy algorithms are not usually optimal → a better planner could reach all interesting properties in less testing steps
  - However becomes computationally more intensive
  - Greedy algorithm works rather well in practice

Copyright © Antti Huima 2004-06. All Rights Reserved.

Formal Conformance Testing 2006

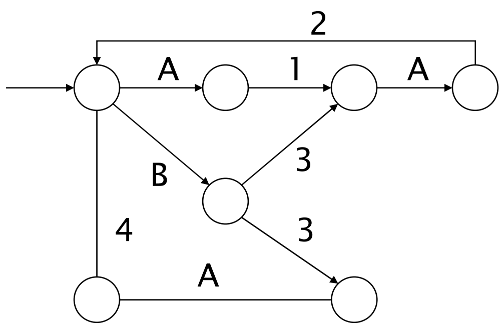
Lecture 12

16th Nov 2006

Implementing a toy FCT tool

- Assume all I/O with system is untimed and has the form of a single stimulus + single response
- Inputs A, B, C, …, outputs 1, 2, 3, …
- Can draw as a state machine

Example

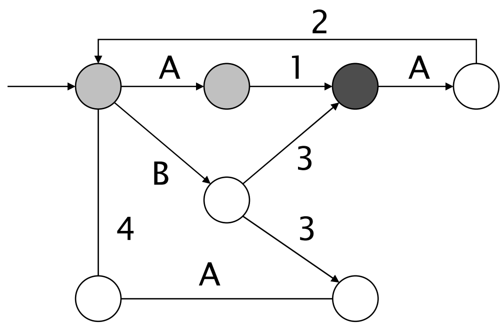


Step 1

- Create a trace inclusion checker
  - Trace e.g. "A1B3C4"
  - Return "pass" if trace found from state chart
  - Return "fail" if trace not in state chart, but every attempt to produce the trace from the state chart fails at a number (output)
  - Return "error" if trace not in state chart, but every attempt to produce the trace from the state chart fails at a letter (input)
  - Otherwise return "confused"

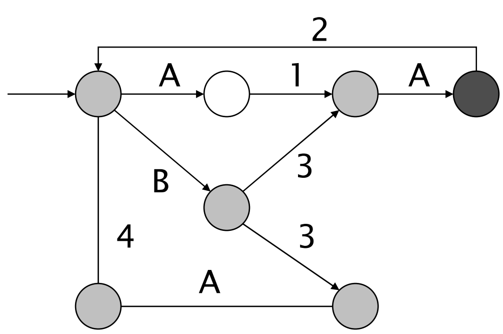
Example

"A1C3"



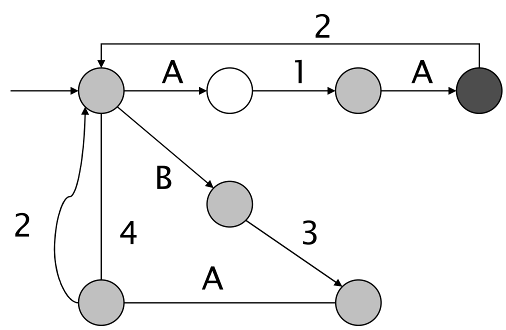
Example

"B3A4"



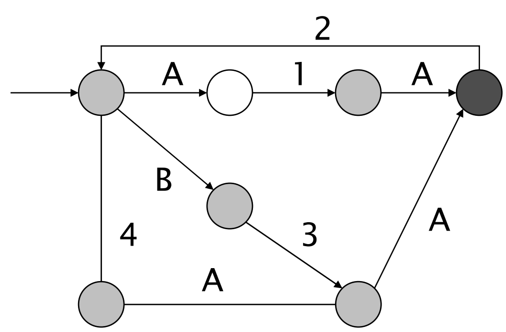
Example

"B3A4"



Example

"B3A4"

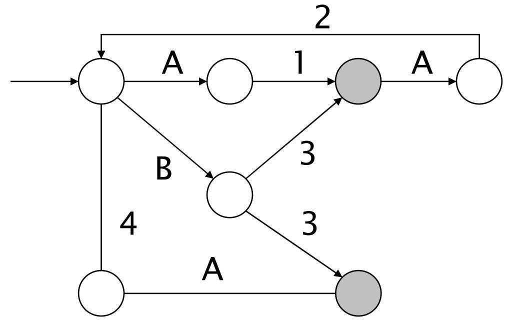


Step 2

- Create a state space explorer that computes for any given "pass" trace the set of those states where the specification state machine can be after the trace

Example

"B3"



Step 3

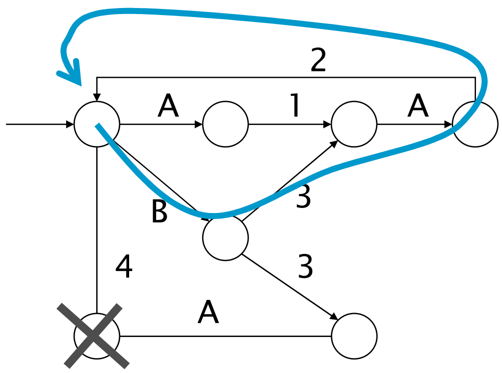
- Build a test execution loop:
  - Check observed trace
  - Compute current specification states
  - Choose an input that is valid in one the states
  - Send it to SUT
  - Receive response
  - Restart

Step 4

- Add testing heuristics
  - Co-operative planning
  - Adversarial planning
- Add test stopping heuristics
  - All states covered
  - “Seems” that no more states can be reached

Example

"B3A2"



Step 5

- Augment the specification / system model with observed transition probabilities from the SUT
- Use these to guide test planning
- Investigate algorithms scalability

Symbolic execution

- If next(s) sets are infinite, the testing algorithms can't be realized "as such"
- Symbolic execution is needed
  - An algorithmic solution to the problem of infinite state sets
  - Well known in general
- For illustration, let us consider the trace inclusion check algorithm

Symbolic trace inclusion check algorithm

```
W := {α[s0]}
V := ∅
While W≠∅
    Choose s from W
    If NotEmpty(s ⊓ LiftTrace(T*))
        Return FOUND
    Else
        W := W – {s}
        V := V ∪ {s}
        N := SymbolicSuccessors(s) ⊓ LiftPrefix(T*)
        W := W ∪ (N – V)
Return NOT FOUND
```

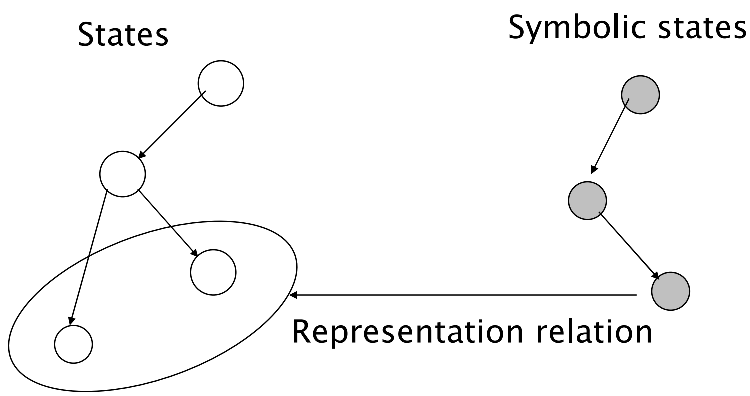
Comments

- α maps a concrete state to a symbolic state representing the singleton set consisting of the concrete state
- ⊓ computes symbolic intersection
- LiftPrefix(T*) returns a symbolic state that represents every state whose trace is either a prefix of T*, or an extension of T*
  - Replaces the check $T \prec T^*$
- LiftTrace(T*) returns a symbolic state that represents every state whose traces is exactly T*
  - Replaces equivalence check
- NotEmpty checks for non-empty symbolic state

Symbolic states

- How symbolic states can be <u>implemented</u>?
- Many techniques known, e.g.
  - BDDs (binary decision diagrams)
  - Constraint systems
    - Linear constraints over reals (→ timed automata)
    - General constraints

Symbolic states



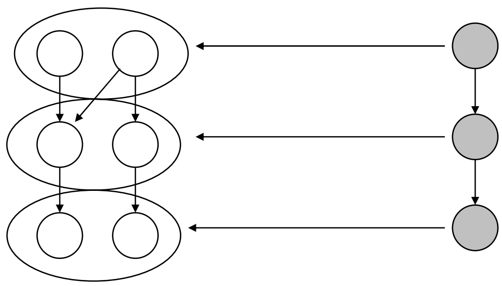
Representation

- Let z by a symbolic state
- γ(z) is a set of states: the set of states <u>represented</u> by z
- For a concrete state s, α(s) is a symbolic state such that γ(α(s)) = {s}

Preliminary Axiom

- If $z \rightarrow z'$, then

  $\gamma(z') \subseteq \{ s' \mid \exists s \in \gamma(z): s \rightarrow s' \}$



Operations for symbolic states

- Emptiness check

  $$\mathrm{Empty}(z) : \gamma(z) = \varnothing$$

- Intersection

  $$\gamma(z \sqcap z') = \gamma(z) \cap \gamma(z')$$

- Subsumption relation

  $$z \sqsubseteq z' \Rightarrow \gamma(z) \subseteq \gamma(z')$$

Symbolic successors

- Next(z) = { z' | z → z' }
- Axiom 1 (completeness):

$$s \in \gamma(z),\ s \to s' \text{ implies}$$
$$\exists z' \in \mathrm{Next}(z) : s' \in \gamma(z')$$

- Axiom 2 (soundness):

$$z' \in \mathrm{Next}(z) \text{ implies}$$
$$\forall s' \in \gamma(z'): \exists s \in \gamma(z): s \to s'$$

Operations needed for symbolic trace inclusion check

- LiftTrace(T)
  - Returns z such that

$$\gamma(z) = \{ s \mid \exists c : s = \langle c, T \rangle \}$$

- LiftPrefix(T)
  - Returns z such that

$$\gamma(z) = \{ s \mid \exists c, T' : s = \langle c, T' \rangle, T' \preccurlyeq T \}$$

Symbolic trace inclusion check algorithm

W := {α[s_0]}
V := ∅
While W≠∅
    Choose z from W
    If not Empty(z ⊓ LiftTrace(T*))
        Return FOUND
    Else
        W := W – {z}
        V := V ∪ {z}
        N := { z'' | z' ∈ Next(z), z'' = z' ⊓ LiftPrefix(T*) }
        W := W ∪ (N – V)
Return NOT FOUND

W = {{s_0}}

Set z contains <c, T> for some c?

Compute successors but filter out states whose traces are not prefixes of T*

Correctness discussion

- Suppose γ(z) are all reachable in the concrete state space
- Suppose z → z'
- Then also γ(z') are all reachable by definition

- On the other hand, suppose s is reachable, and z is reachable such that γ(z) contains s
- Suppose s → s'
- Then z' exists in the set Next(z) such that s' ∈ γ(z)

Discussion

- The symbolic state space depicts the whole infinite state space, but can be in itself finite as a structure
- Requires good way to actually represent and manipulate symbolic states

Constraint solutions

- Constraint set: {X1>0, X3=X1+0.1, number X2, X4=X2+1}
- X1=0.2, X3=0.3, X2=9, X4=10 is a solution
- Corresponds to a real execution
- X1=-1 does not lead to a solution
  - Negative time stamp!
- X1=1, X3=10 does not lead to a solution
  - Wrong wait time!
- X2="hello" does not lead to a solution
  - Received value not number!

Constraint sets

- Constraint set = system of equations over data
- E.g.

$$x < y$$
$$x*z = 9$$
$$s = \text{“foo”}$$
$$a = \mathrm{append}(s, s)$$

Computational point of view

- Constraint sets are easy to create, <u>difficult</u> to solve
- Unsolvable problems abound
- But many realistic cases can be handled

Using constraints

- System state structure <c, T>
- Assume that <c, T> is otherwise concrete represented, but that c and T can mention constraint variables
- Add a constraint set
- Symbolic state is of the form <<c, T>, C> where C is a constraint set
- Constraint set constraints the values of the constraint variables
- A concrete state is represented iff it is obtained by replacing the constraint variables with a solution of the constraint set

Example

- c = [t→X1, x→X2, …]
- T = $<\{<X2_{in}, X1>, <X4_{out}, X3>\}, X3>$
- C = {X1>0, X3=X1+0.1, number X2, X4=X2+1}
- <<c, T>, C> is a symbolic state

Intersections

- We assume the symbolic states are structured so that if z and z' represent at least one concrete same state, there is 1-1 correspondence between constraint variables of the symbolic states
- This can be provided

Intersections ctd

- We can then take two symbolic states $z=\langle\langle c, T\rangle, C\rangle$ and $z'=\langle\langle c', T'\rangle, C'\rangle$ and proceed to compute their intersection
- Map all constraint variables of z' to those of z, with mapping Q (if not possible, intersection empty)
- Intersection is $\langle\langle c, T\rangle, C \wedge Q(C')\rangle$
- Assumes constraint sets are closed under conjunction

Copyright © Antti Huima 2004-06. All Rights Reserved.

Intersections ctd

- To make LiftTrace, LiftPrefix work, we must also allow for a case where the control part is undefined
- $\langle\langle c, T\rangle, C\rangle \sqcap \langle\langle ?, T'\rangle, C'\rangle$: match T against T', then yield

$$\langle\langle c, T\rangle, C \wedge Q(C')\rangle$$

- (or empty symbolic state)

Emptiness check

- Emptiness check can be now reduced to checking for the satisfiability of a constraint set

Subsumption check

- Subsumption check can be reduced now to checking that a constraint set implies another one
- To check for $C \Rightarrow C'$, check for the satisfiability of $C \wedge \neg C'$
- Assumes now that constraint sets are closed also under negation → full Boolean closure

Where constraint variables come from?

- There are two causes for constraint variables in symbolic execution:
  - Internal choices (e.g. (random))
  - Input from environment (message, timeout)
- But these two cases are completely different!
  - Internal choices and input from environment correspond to decisions made by distinct parties (SUT, Tester)
  - A problem lurks…

Alternating quantifiers!

- Basically, we would like to create testing plans that cover <u>all</u> potential internal choices of a correctly working SUT
- This yields to constraint solving over alternating quantifies (→ adversarial planning)
- Seems to be computationally infeasible
- Must straighten some curves, and assume a co-operative SUT
- With a co-operative SUT, SUT choices and Tester choices are on par

More algorithms

- The symbolic versions of the full testing algorithms are left as an exercise for the student

Formal Conformance Testing 2006

LAST LECTURES

30th Nov 2006

EXAMINATION INFO

- Next examination 15th December
- For those who leave from country at New Year, there is possibility for taking the examination orally <u>next Monday</u>
  - The results will be calibrated with the results from the written exam before being recorded

Topics today

- The classic IOCO theory
- Critique of IOCO

Ioco theory

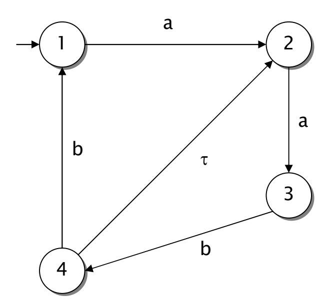
- The "classic theory"
- Often referred to as the "ioco" testing theory and is quite well known among the academic peoples
- A framework developed by Tretmans, Heerink et al.
- Dates to early 90's

loco theory overview

- LTSs (labeled transition systems) = finite state machines
- No notion or only a very weak notion of time
- Some tools have been developed based on the theory, for example TorX

Labeled transition systems

- A labeled transition system is a tuple $<S, L, T, s_0>$ where
  - S is the set of states
  - L the set of transition labels
  - $T \subseteq S \times L_\tau \times S$ the transition relation (with $L_\tau = L \cup \{\tau\}$)
  - $s_0 \in S$ the initial state.

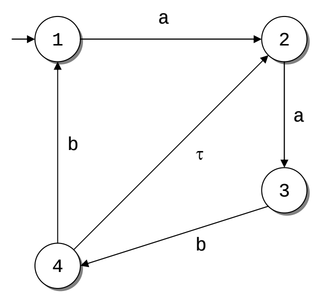


S={1,2,3,4}
L={a,b}
T={<1,a,2>,<2,a,3>,<3,b,4>, <4,τ,2>,<4,b,1>}
S_0=1

Traces

- The traces of an LTS are obtained by "walking" in it starting from the initial state, and collecting all symbols except τ's which denote "silent activity" and which are removed.

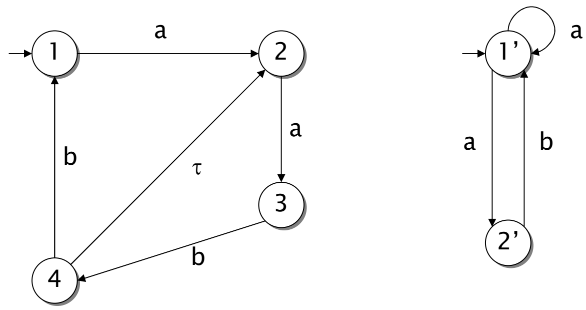
ε	a	aa
aab	aabb	aaba
aabba	aabab	aababa



Parallel composition

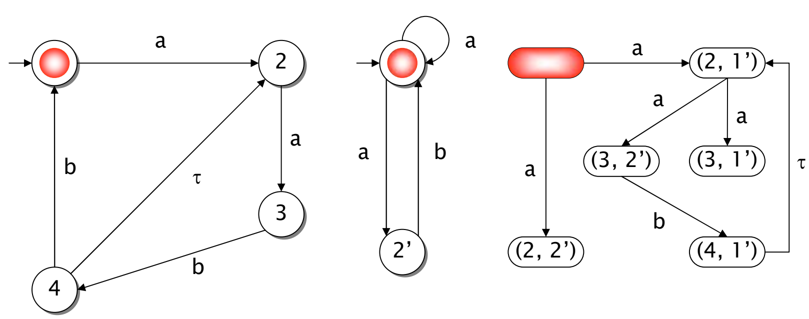
- The parallel composition of two LTSs is traditionally denoted by $L \parallel L'$.
- This construct creates a new LTS from two LTSs.
- Two LTSs run in synchrony, always taking arcs together with same labels. An exception is the τ-label which is not synchronized.
- This synchronization is not directional but completely symmetric.
  - Can be therefore called a "handshake".

Example



- There are eight state pairs in total. So the parallel composition will have eight or less states. It is so small that we can construct it explicitly.

Example



- The resulting LTS has only six states. The reason is that the states <1, 2'> and <4, 2'> are not reachable.
- The second LTS does not allow for two b's in a row.

More on the parallel composition

- Parallel composition models "synchronous, symmetric communication" or "symmetric handshake".
- Powerful construct: the reachability problem (= can a given composite state be reached) for parallel composed LTSs is PSPACE-complete (on the number of composed LTSs). This means that the problem is very hard.
- In the ioco testing theory, parallel composition is used to model the communication between Tester and the SUT (both are assumed to be LTSs).

Parallel composition and realistic I/O

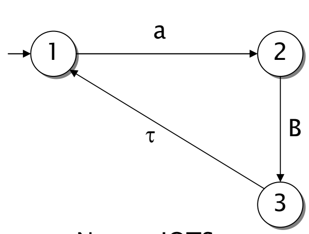
- In parallel composition, the two LTSs can take step with label a ($\neq \tau$) only if they do that together.
- This means that if a models, say, a message from Tester to SUT, then the SUT can refuse to receive the message (just by not having an outgoing transition with the label a).
- This is disturbing, because after all it is in the Tester's discretion to decide when to send messages and when not.
- These aspects lead us to the concept of an IOTS.

IOTS

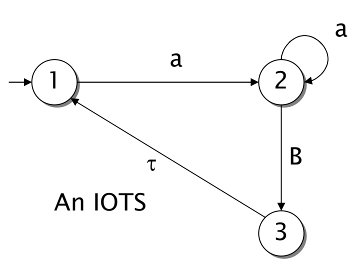
- IOTS = Input Output Transition System.
- The set of labels L is partitioned into input labels L_I and output labels L_O.
- An IOTS is a standard LTS that has the following extra property:
- For every reachable state s in the LTS, there exists a path from s that accepts any arbitrary input label first. This means that <u>you cannot refuse an input and that you can't deadlock.</u>

Example

- Assume the set of input labels is {a} and the set of output labels is {B}.



Not an IOTS



An IOTS

Testing Theory for IOTSs

- In the "ioco" testing theory, the Tester and the SUT are assumed to be IOTSs.
- Obviously, the Tester and SUT are mirror images of each other in the sense that outputs from SUT are inputs to Tester and vice versa.
- Hence, if L_O is the set of outputs from SUT, then this is the set of inputs to Tester, which must be always enabled in Tester.
- The specification is also an IOTS. (Actually, it can be a non-IOTS LTS—the theory speaks of "partial specifications".)

The core idea

- Assume we have some definition of "observations" that an LTS produces; we denote this for now by obs(L) for an LTS L.
- Given a tester t, SUT i and specification s, let us say that <u>t confirms i w.r.t. s</u> if

  $$\mathrm{obs}(t \parallel i) \subseteq \mathrm{obs}(t \parallel s).$$

  (All the three entities are IOTSs).
- We can now say that an implementation i <u>conforms to a specification</u> s if all possible testers confirm i w.r.t. s.
- What are the observations?

Basic Observations

- We assume that the observations that we can make of an LTS L are the following:
  - The set of all traces of L, plus
  - the set of those traces of L after which L can be in a deadlock
- Now write obs(L) $\subseteq$ obs(L') if the subset relation holds for both the sets mentioned above.
- This leads to the input-output testing relation $\leq_{iot}$. We write i $\leq_{iot}$ s to denote that i conforms to s in this sense.

Input-output testing relation

- When an implementation conforms to a specification in the sense of $\leq_{iot}$…
  - If you <u>can</u> produce a trace against the implementation, then you could produce the same trace against the specification (= reference implementation) (but not necessarily vice versa).
  - If you <u>can</u> bring the implementation into a state where it just waits for input, then you could do the same with the specification (but not necessarily vice versa).

Alternative formulation

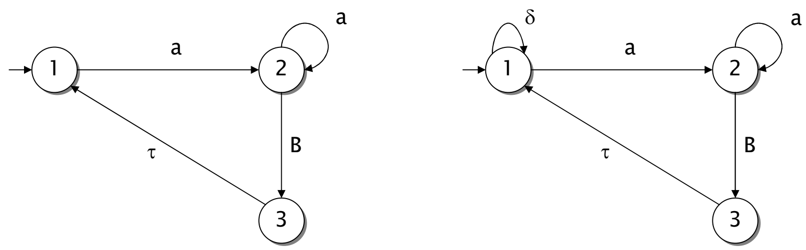
- An alternative way to define the same result is given next.
- $i \leq_{iot} s$ iff
  $traces(i) \subseteq traces(s)$ and $Qtraces(i) \subseteq Qtraces(s)$
  where Qtraces(L) is the set of those traces of L after which L can be in a state where only transitions labeled by inputs are possible (i.e. L is waiting for input and cannot proceed without one; a "quiescent state"—hence 'Qtraces').
- So, we see here a standard trace inclusion problem… at least almost. Note that Tester is not mentioned!

Quiescence…

- Quiescence traces model the assumption that we can detect when the SUT is not going to anything observable before it gets more input.
- Ultimately, this complication comes from the fact that there is no time in the theory.
- But actually there exists a stronger variant of this idea.

Repetitive Quiescence

- Let us assume that we patch the SUT so that whenever it is just waiting for input, it <u>can</u> send out a meta-message δ which denotes "I'm waiting for input" or "I'm quiescent".



Repetitive Quiescence (ctd)

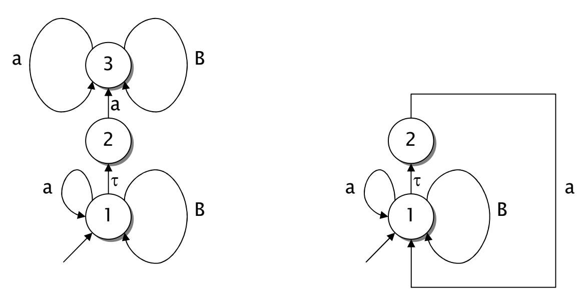
- The name for δ is "suspension".
- We call the traces of an IOTS with this extension (can produce δ when no output is possible) "suspension traces", denoted by Straces(L).

Ioco relation

- Now an implementation i conforms to a specification s iff Straces(*i*) $\subseteq$ Straces(*s*).
- This corresponds to the inclusion of observations by all testers who can observe I/O behavior, deadlocks and δs.
- This is the ioco testing relation.

What is the Difference?

- $\leq_{iot}$ is based on the possibility of detecting lack of output after a test run, but only at the end of a test run.
- In ioco it is possible to detect quiescence also in the midst of a test run.



General comments

- Ioco theory is low-level theory
  - Pragmatic systems are not given as LTSs but as Java programs, UML state charts, …
  - Not a problem but a statement about the focus of the theory
- In principle no need to assume finite LTSes
  - But in the practice, algorithms focus on finite LTSes

Finite LTSes

- Usually finite LTSes are assumed in the context of ioco
- But realistic systems usually have infinite or very big state graphs
- Leads to the need to do <u>manual abstraction</u>

Manual abstraction in testing

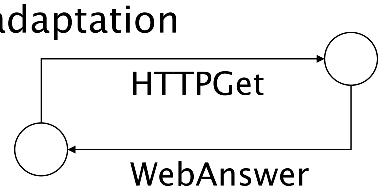
- How to create a small finite state machine (i.e. LTS) from a specification generating a big/infinite state space?
- Drop out details
- Replace data with abstract placeholders

Benefits

- Resulting small state machines are easy to manipulate algorithmically
  - All kinds of interesting analyses and constructs are possible
- Strengthened focus on abstract control structure

Cons

- Driving real testing with abstract inputs can be impossible or very difficult—the system under test wants concrete input
  - Complicated extra adaptation component



Relevance of ioco theory

- A common framework
  - Many articles written
- Main contributions
  - Link the general practice of conformance testing (from telecom domain) with formal methods
  - Establish the flourishing study of formal models based conformance testing

Formal conformance testing and software process

- How can formal conformance testing be integrated into a software process?
- Main challenges
  - Where get executable/formal specification or design?
  - Where to get a tool?
  - What kind of process support is needed?

Specification?

- Clearly, a formal specification does not need to be in greek
- But it must have well-defined meaning
- In our context, it should be an executable reference design (e.g. in Scheme)
- Where to get it?

How to get a reference implementation?

- First do reference implementation, then implement the real system using it as a guide
- Reverse-engineer from the implementation afterwards
- Develop at the same time as the real implementation, based on same system requirements
- Create reference implementation / system model, code-generate real system from it (→ model driven architecture)

Tool support?

- Only emerging
- Main challenges
  - Algorithmic complexity
  - Conceptual difficulty
  - Usability
  - Business case

Process support

- Specifications (executable reference implementations) are software artifacts!
  - They need a software process themselves
  - Testing!
  - Validation!

CONCLUSIONS

- FCT / MBT is becoming a mainstream technology
- Realistic MBT implementation requires some advanced machinery
- The main idea is to derive tests from system specifications
- You were a great audience! ☺