Testing of Concurrent Systems 2004

Lecture 1
14th Sep 2004

Welcome!

- This is T-79.190: Testing of Concurrent Systems
- Lectures from 8 to 10 am, tutorials from 10 to 11 every Tuesday at T3
- Cancellations and other notes at the web page (go to http://www.tcs.hut.fi/)

Lecturer

- Antti Huima = me
- "Special teacher" = not member of HUT staff
- Work = Vice President of Research and Development at Conformiq Software

Practical Matters

- Website contains all important information
- The news group can be used for discussion, but I will not follow it
- Lecture notes will be available on the web for "early access"
- Printed and distributed via the lecture notes print house

Requirements

- Pass the course = pass the examination
- Tutorials do not award extra points
- Tutorials form part of the requirements for the examination
- Model answers will be made available

Tutorials

- Tutorials from 10 to 11 am, after the lectures
- Begin next week
- No tutorial today
- Subject = lectures of the previous week

Subject

- Real subject = formal conformance testing (FCT)
- "Testing of concurrent systems" is a historical title
- What is "formal conformance testing"?

Testing

- Testing is the process of
  1. interacting with a system, and
  2. evaluating the results, in order to
  3. determine if the system conforms to its specification
- In testing setup, the system is known as the system under test (SUT)

Interacting

- If you can't interact with a system, the system is uninteresting
- Interacting covers anything you can do with the system

Conforms

- Interaction does not imply judgement
- Conformance = correspondence in form or appearance
- Conformance to a specification = "works as specified"
- Were the results of the interaction allowed by the specification?

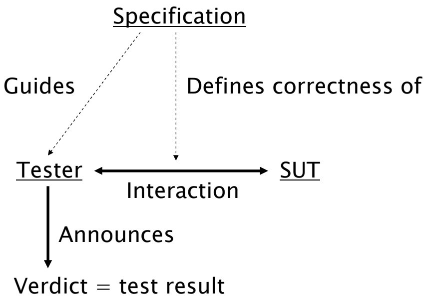
Formal

- Formal = "according to a form"
- Here: testing is based on a mathematical, formalized foundation
- Not: testing based on a rigorous process where you need to fill lots of bureaucratic forms
- Also: "formal methods based", but this is very vague

Operational specification

- Specifies how a system should work
- Operational = describes behaviour, not e.g. usability scores
- Operational: "after 3 s, system must respond with X"
- Non-operational: "users must like the stuff"
- From now on just "specification" (assume operational)

FCT setup



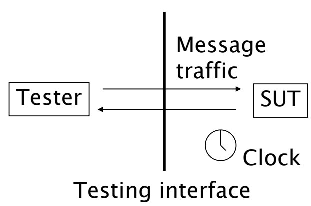
Tester

- Tester has two functions:
  - Interact = generate behaviour
  - Give verdict = judge behaviour
- These two functions can be separated

Verdicts

- Typical verdicts:
  - PASS = system behaved ok
  - FAIL = system behaved badly
  - ERROR = tester messed up
  - INCONCLUSIVE = PASS, but some important feature was not yet tested

Testing interface



Testing interface

- All interaction happens through the testing interface
- Bidirectional message passing
- All transmissions have a time stamp
- Every event has a distinct time stamp

Directions

- Input
  - input to the SUT
  - output from the tester
- Output
  - output from the SUT
  - input to the tester

Alphabets

- Σ_{in} is the set of input messages
- Σ_{out} is the set of out messages
- Σ is the union of the two
- Messages “contain” their direction
- Alphabet = traditional name for a set of potential messages

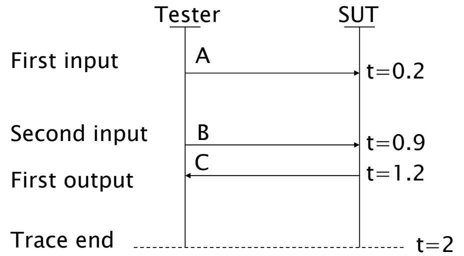
Events

- Event = message + a time stamp
- Thus, event = (member of Σ) + (nonnegative real number)
- Formally, set of events is $\Sigma \times [0,\infty)$
- E.g. <“hello world”$_{in}$, 1.4 s>

Traces

- A trace denotes an observation of events for a certain time
- Trace = a finite set of events with distinct time stamps + end time stamp
- E.g. <{<“hello”$_{in,}$ 0.5>}, 0.8>

Graphical sketch



<{<A,0.2>,<B,0.9>,<C,1.2>},2>

lecture 1 summary

- Testing = interact + judge
- Specification, tester, SUT
- Testing interface = point of interaction
- Trace = a finite series of events observed during a finite time span

Testing of Concurrent Systems 2004

Lecture 2

14th Sep 2004

Review of previous lecture

- Testing = interact + judge
- Specification, tester, SUT
- Testing interface = point of interaction
- Trace = a finite series of events observed during a finite time span

Process notation

- We need a notation for "computational processes", i.e. a programming language to describe
  - implementations = SUTs
  - operational specifications as "reference implementations"
  - full testers
  - testing strategies = interaction strategies

Requirements

- Support data, time, concurrency
- Familiar
- Compact
- Executable

The choice

- Scheme, a dialect of LISP
- But standard Scheme lacks concurrency and time
- Solution: extend Scheme slightly

Introduction to Scheme

- Scheme is a easy and clean dialect of LISP
- Scheme = interpreter for applicative order λ-calculus
- See e.g. Abelson & Sussman

Standard Scheme

```
(define (factorial n)
  (if (<= n 1)
      1
      (* n (factorial (- n 1)))))
```

Standard Scheme

```
(define (make-add-to-list n)
  (lambda (ls)
     (map (lambda (x) (+ x n)) ls)))
(let ((z (make-add-to-list 5)))
  (z '(1 2 3)))

→ '(6 7 8)
```


Extensions

- We extend Scheme with procedures:
  - spawn
  - make-rendezvous-point
- A special form:
  - sync

Spawn

- Creates a new thread
- Use: (spawn <thunk>)
- Returns nothing

```
(spawn (lambda () ...))
```


Make-rendezvous-point

- Creates a point of synchronous internal communication
- Use: (make-rendezvous-point)
- Returns a new rendezvous point
- Rendezvous points are used by the (sync ...) form

Sync

- General I/O and wait form
- Use:

```
(sync
  (input <var> <body> ...) ...
  (output <expr> <body> ...) ...
  (read <point> <var> <body> ...) ...
  (write <point> <expr> <body> ...) ...
  (wait <expr> <body> ...) ...)
```


External I/O

- (input <var> <body> ...) attempts to read a message from the environment; if successful, store data to <var> and continue with <body> ...
- (output <expr> <body> ...) attempts to write a message to the environment; if successful, continue with <body> ...

Internal I/O

- (read <point> <var> <body> ...) attempts to read a message from point <point>; if successful, store it to <var> and continue with <body> ...
- (write <point> <expr> <body> ...) attempts to write <expr> to point <point>; if successful, continue with <body> ...

Timeout

- (wait <expr> <body> ...) attempts to wait for <expr> seconds; if nothing else happens until that amount of time, continue with <body> ...

Choice

- Choice between all items enabled at the same point of time is nondeterministic

Examples

```
(define (run)
  (sync (input x (run))))

(run)
```


Zero-time execution principle

- We assume that all Scheme execution consumes zero time
- The only exception is waiting at sync

Examples

```
(define (echo)
  (sync (input x
            (sync (output x (echo))))))

(echo)
```

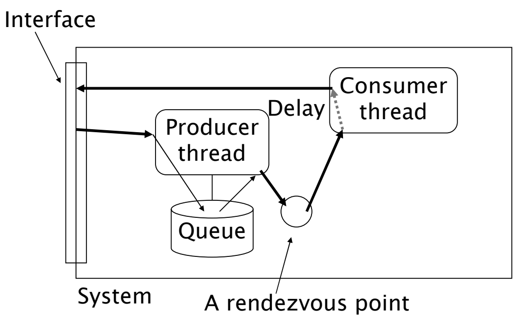
Examples

```
(define (echo)
  (sync (input x (wait-and-echo x))))

(define (wait-and-echo x)
  (sync (wait (a-delay)
            (sync (output x (echo))))))

(echo)
```


Queued echo

```
(define (run)
  (let ((queue (make-queue))
        (point (make-rendezvous-point)))
      (spawn (lambda ()
                  (producer point queue)))
      (spawn (lambda ()
                  (consumer point)))))

(run)
```


Queued echo architecture



Queued echo

```
(define (consumer point)
  (sync (read point x
          (sync (wait (a-delay)
                   (sync output x
                     (consumer
                       point))))))))
```


Queued echo

```
(define (producer point queue)
  (if (empty-queue? queue)
      (sync (input x (queue-insert! queue x)))
      (sync (input x (queue-insert! queue x)))
            (write point (queue-front queue)
                         (queue-remove! queue))))
  (producer point queue))
```


Lecture 2 summary

- We use Scheme with concurrency extensions to denote processes
  - spawn
  - make-rendezvous-point
  - sync

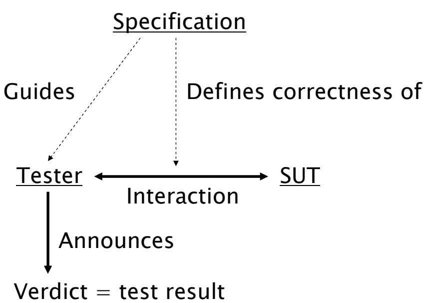
Testing of Concurrent Systems 2004

Lecture 3
21th Sep 2004

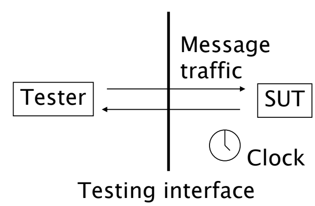
Course this far

1	▸ Introduction ▸ General concepts ▸ Traces
2	▸ Concurrent Scheme

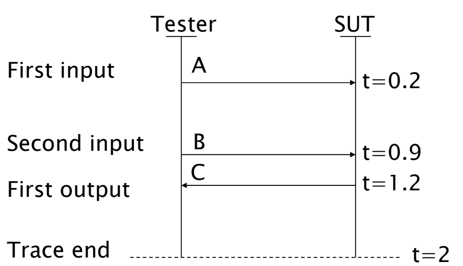
FCT setup (replay)



Testing interface (replay)



A trace (replay)



<{<A,0.2>,<B,0.9>,<C,1.2>},2>

Traces

- Traces denote finitely long observations on the testing interface
- A trace contains a finite number of events and an end time stamp
- Traces are the *lingua franca* for discussing behaviours

Traces

- Alphabet
- Event
- Trace
- Trace prefix
- Empty trace
- Trace extension
- Snapshot
- Difference time

Alphabets

- Σ_{in} is the set of input messages
- Σ_{out} is the set of out messages
- Σ is the union of the two
- Messages "contain" their direction
- Alphabet = traditional name for a set of potential messages

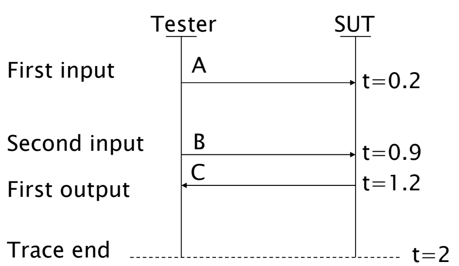
Events

- Event = message + a time stamp
- Thus, event = (member of Σ) + (nonnegative real number)
- Formally, set of events is $\Sigma \times [0,\infty)$
- E.g. <"hello world"$_{in}$, 1.4 s>

Traces

- A trace denotes an observation of events for a certain time
- Trace = a finite set of events with distinct time stamps + end time stamp
- E.g. <{<"hello"$_{in,}$ 0.5>}, 0.8>

Graphical sketch

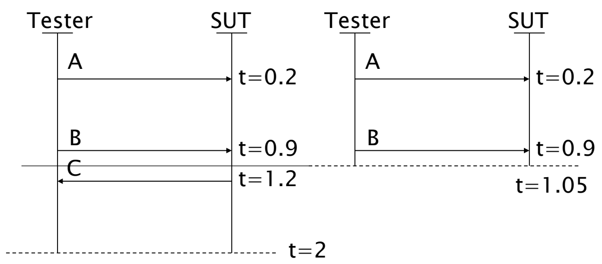


<{<A,0.2>,<B,0.9>,<C,1.2>},2>

Prefixes

- A trace is a prefix of another, if the first trace can be extended in time to become the second one
- Let T and T' be traces
- $T=\langle E,t\rangle$ is a prefix of $T'=\langle E',t'\rangle$ (write $T \preccurlyeq T'$) if
  - $t \leq t'$ and
  - $E = \{ \langle\alpha, \kappa\rangle \mid \langle\alpha, \kappa\rangle \in E' \wedge \kappa < t \}$

Prefix sketch



Empty trace

- $<\emptyset,0>$ is the empty trace, denoted by ϵ
- The empty trace is a prefix of every other trace
- Empty trace has no information content

Extensions

- A trace T is an extension of trace T' if the trace T' is a prefix of trace T
- Thus, being extension = reverse of being prefix

Prefix set

- Pfx(T) is the set of all prefixes of T
- $\text{Pfx}(T) = \{ T' \mid T' \preccurlyeq T \}$
- Note: If $T \preccurlyeq T'$ then $\text{Pfx}(T) \subseteq \text{Pfx}(T')$

Snapshot

- Denote $\Sigma_\tau = \Sigma \cup \{\tau\}$
- Here τ is an object that does not belong to set Σ
- Let $T = <E,t>$
- Assume $\kappa < t$
- Denote by $T|_\kappa$ the event at time κ, or τ if no event at trace T has time stamp κ

Snapshot example

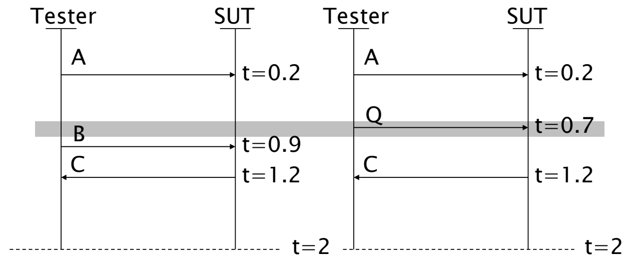
- Suppose $T = <\{<A,1>\}, 2>$
- $T|_1 = A$
- $T|_{1.5} = \tau$
- $T|_2$ is not defined
- $T|_3$ is not defined

Difference time

- Suppose T and T' are traces such that T is not a prefix T' and T' is not a prefix of T
- T and T' are hence not equal
- Define

$$\Delta(T,T') = \min t^* : T|_{t^*} \neq T'|_{t^*}$$

Difference time sketch



Specifications

- Set of specifications
- Set of valid traces
- Prefix-completeness
- Seriality

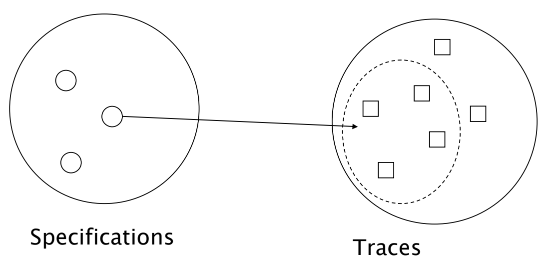
Set of specification

- $\mathbb{S}$ is a countable set of specifications
- Could be e.g.
  - Set of syntactically correct UML state charts
  - Set of valid English documents
- Structure not relevant
- Assume exists

Valid traces

- Every specification denotes a set of traces: the set of *valid traces*
- If S is a specification, Tr(S) is the set of valid traces for S
- Tr(S) must contain ϵ
- Tr(S) must be *prefix-complete*
- Tr(S) must be *serial*

Specifications



Prefix-completeness

- A set X of traces is prefix-complete if the following holds:
- If T $\in$ X and T’ $\preccurlyeq$ T then also T’ $\in$ X
- If a trace belongs to a prefix-complete set, then also all its prefixes belong to the set
- Why Tr(S) must be prefix-complete?

Motivation for prefix-completeness

- Tr(S) denotes a set of *acceptable behaviours*
- Assume T is an acceptable behaviour
- Can you imagine a case where T’, a prefix of T, would be not acceptable?

Testing of Concurrent Systems 2004

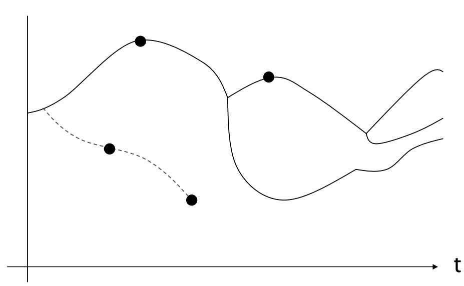
Lecture 4

21th Sep 2004

Seriality

- A set X of traces is *serial* if for every <E, t> $\in$ X and for every $\delta > 0$ this holds:
- There exists <E’, t+δ> $\in$ X such that <E, t> $\preccurlyeq$ <E’, t+δ>
- Every trace of X has at least one arbitrarily long extension in X

Seriality sketch



Motivation for seriality

- Suppose non-serial Tr(S)
- There exists a valid trace T without an extension
- Let T’ $\preccurlyeq$ T
- Is the behaviour T’ acceptable?
- Why? And why not?

Implementations

- We assume there exists a countable set of implementations, denoted by $\mathbb{I}$
- Could be e.g.
  - Set of all valid JAVA programs
  - Set of all valid C programs
  - Set of all functioning digital circuits

Failure model

- Failure model links a specification to its potential implementations
- A failure model is a function

$$\mu: \mathbb{S} \to (\mathbb{I} \to [0,1])$$

- For every $s \in \mathbb{S}$, it holds that

$$\Sigma_i\ \mu(s)[i] = 1$$

- Hence $\mu(s)$ is a discrete probability distribution over implementations

Use of failure models

- Failure model is a hypothesis about implementations and their potential defects
- Example: Boundary Value Pattern and the related failure model

Testing strategies

- A testing strategy is a strategy on how to interact with an implementation
- Let $\mathbb{T}$ denote the countable set of all testing strategies
- What happens when a testing strategy is executed "against" an implementation?

Execution

- Testing strategy + implementation yields a sequence of traces T_1, T_2, T_3,…
- Here $T_1 \preccurlyeq T_2 \preccurlyeq T_3 \preccurlyeq \ldots$
- These correspond to *test steps*
- Many different trace sequences are possible
- How do we formalize this?

Semantic function ξ

- Maps implementation, testing strategy and "system state" to extensions of the currently observed trace
- Actually to a probability distribution of extensions
- System state = trace observed this far

ξ function properties

- Gives a probability distribution
- Test steps are proper trace extensions
- Progressivity
- Test steps are disjoint

Signature

- Let $\mathcal{T}$ denote the set of all traces
- The signature is

$$\xi : \mathbb{I} \times \mathbb{T} \times \mathcal{T} \to (\mathcal{T} \to [0, 1])$$

Gives probability distribution

- For all i, s and T, it must hold that

$$\Sigma_{T'}\, \xi(i,s,T)[T'] = 1$$

Test steps = proper trace extensions

- For all i, s, T and T' it must hold that

$$\xi(i,s,T)[T'] > 0 \Rightarrow T \prec T'$$

- Hence: every test step consumes time

Test step disjointness

- For any i, s, T, and T_1 and T_2 it must hold that if $T_1 \neq T_2$ and
- $\xi(i, s, T)[T_1] > 0$ and
- $\xi(i, s, T)[T_2] > 0$, then
- $T_1 \not\prec T_2$, and
- $T_2 \not\prec T_1$
- A technical convenience

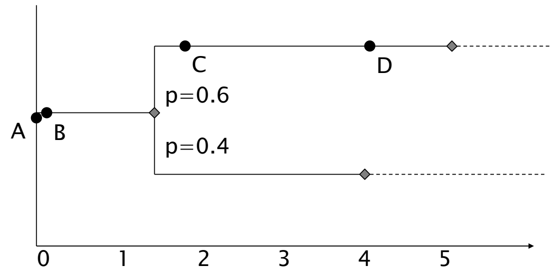
Progressivity

- There does not exist an infinite sequence T_1, T_2, T_3, … and a constant $K \in \mathbb{R}$ such that
- $\xi(i, s, T_i)[T_{i+1}] > 0$ for all i, but such that for all $T_i = <E_i, t_i>$ it holds that $t_i < K$.

Trace probabilities

- Let P[i, s, T] denote the probability of observing T as a prefix of a long enough trace when strategy s is executed against implementation i
- Idea is to compute the product of the preceding test step probabilities
- Multiply this with the probabilities of those test steps that produce extensions of trace T
- Technical definitions in the handouts

Execution sketch



Testing of Concurrent Systems 2004

Lecture 5

28th Sep 2004

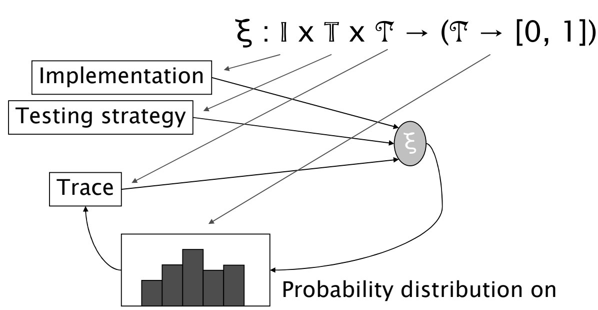
Course this far

14.9	1	▸ Introduction ▸ General concepts ▸ Traces
	2	▸ Concurrent Scheme
21.9	3	▸ Traces, specifications
	4	▸ Seriality, execution introduction

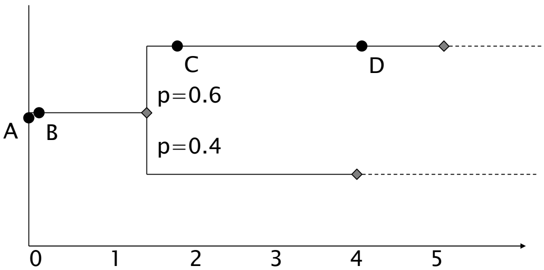
Traces and specifications

- Trace = set of events + end time stamp
  - Event = message + time stamp
  - Prefix, extension, snapshot
- Specification ≅ set of valid traces
  - Prefix-closed
  - Serial

Execution

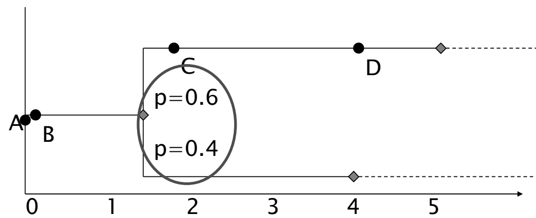


Execution sketch



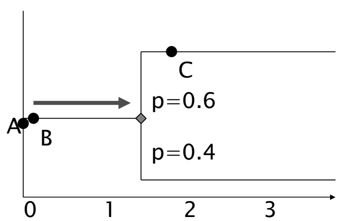
Gives probability distribution

- For all i, s and T, it must hold that $\Sigma_{T'}\,\xi(i,s,T)[T'] = 1$



Test steps = proper trace extensions

- For all i, s, T and T' it must hold that
  $\xi(i,s,T)[T'] > 0$
  $\Rightarrow T \prec T'$
- Hence: every test step consumes time



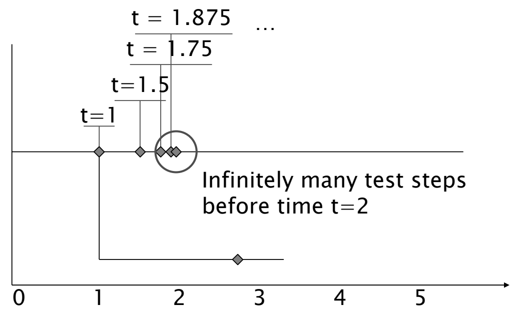
Test step disjointness

- For any i, s, T, and T_1 and T_2 it must hold that if $T_1 \neq T_2$ and
- $\xi(i, s, T)[T_1] > 0$ and
- $\xi(i, s, T)[T_2] > 0$, then
- $T_1 \nprec T_2$, and
- $T_2 \nprec T_1$
- A technical convenience

Progressivity

- There does not exist an infinite sequence $T_1, T_2, T_3, \ldots$ and a constant $K \in \mathbb{R}$ such that
- $\xi(i, s, T_i)[T_{i+1}] > 0$ for all i, but such that for all $T_i = \langle E_i, t_i \rangle$ it holds that $t_i < K$.

Non-progressivity sketch



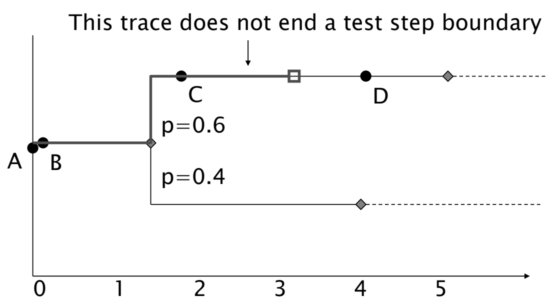
Choice of ξ

- We have defined properties of ξ, not the function itself
- The particular choice for ξ depends on
  - the set of implementations $\mathbb{I}$,
  - the set of testing strategies $\mathbb{T}$, and
  - the desired structure of test steps.

Trace probabilities

- Random experiment:
- An implementation i and a testing strategy s have been chosen
- A trace prefix T* has been fixed, T*=<E,K>
- s is executed against i many times, yielding traces $T_1=\langle E_1,t_1\rangle$, $T_2=\langle E_2,t_2\rangle$, …, such that for all n, $t_n > K$
- What is the probability that for a uniformly chosen n, $T_n[K] = T^*$?
  - X[t] is that prefix of X whose end time stamp is t

Problem



Solution

- Traces that end at test step boundaries are easy: compute product probability
- Traces that end at non-boundaries require an extra construct

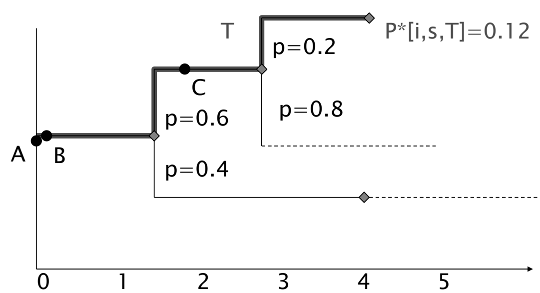
Step 1: traces at test step boundaries

- Denote by P*[i,s,T]:

$$\max_{T_1,\ldots,T_n}: \prod_{i\in[1,\, n-1]} \xi(i,s,T_i)[T_{i+1}]$$
$$\text{where } T_1=\epsilon \text{ and } T_n=T$$

- P*[i,s,T] is the compound probability for trace T, if T "happens" at test step boundary

Sketch



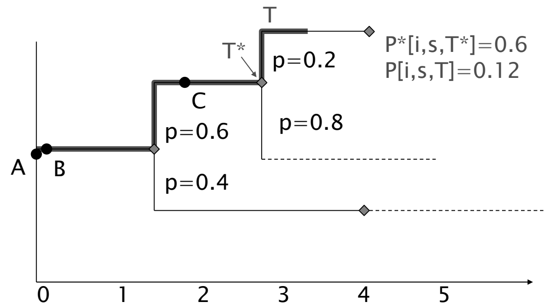
Step 2: traces at non-boundaries

- Denote by P[i,s,T]

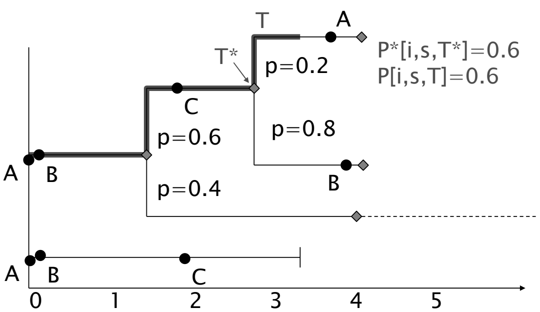
$$P^*[i,s,T^*] \times \left(\sum_{T':T\preccurlyeq T'} \xi(i,s,T^*)[T']\right)$$

- T* is largest prefix of T such that P*[i,s,T*] > 0

Sketch



Sketch



Sanity checks

- If P*[i,s,T] > 0,
  - then P[i,s,T] = P*[i,s,T].
  - Ok.
- If P[i,s,T] = 0 (trace T cannot be produced),
  - there still exists the greatest prefix T* of T such that P*[i,s,T*] > 0.
  - Every test step succeeding T* must result in a trace differing from T — ok.

Sanity check 1 memo

- Assume P*[i,s,T]>0
- Note T* = T
- P[i,s,T] =

$$P^*[i,s,T] \times \left(\sum_{T':T\preccurlyeq T'} \xi(i,s,T)[T']\right)$$

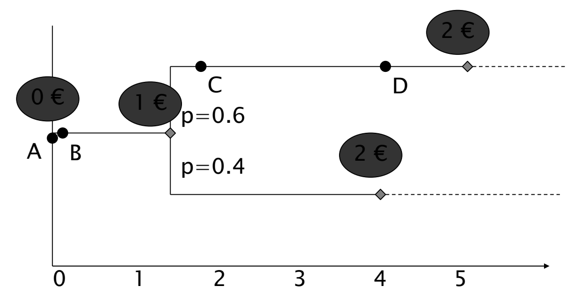
Execution summary

- ξ defines execution semantics
- Properties for ξ
  - Gives probability distribution over traces
  - Test step = trace extension
  - Test step disjointness
  - Progressivity
- However, no concrete structure
- P[i,s,T] is the probability of producing trace T when s is run against i
  - Hides test steps

Why test steps?

- 1 test step =
  - unit of testing cost
  - unit of benefit
- Testing can be stopped between test steps, but not during them
  → stopping criteria
- Technical construct for describing arbitrarily long executions without the concept of “an infinite trace” (there is no such concept here)

Cost or benefit



Measuring the “size” of a trace

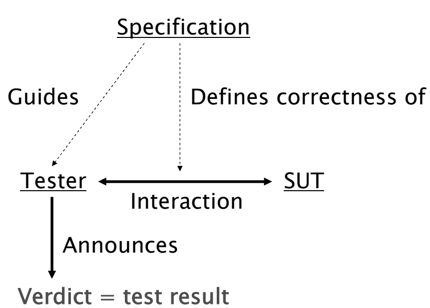
- Temporal length = end time stamp
- Size of event set
- Number of test steps used to produce

Testing of Concurrent Systems 2004

Lecture 6

28th Sep 2004

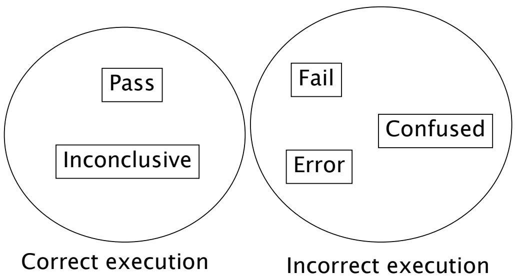
Verdicts



Verdicts

- Pass
- Fail
- Error
- Inconclusive
- Confused

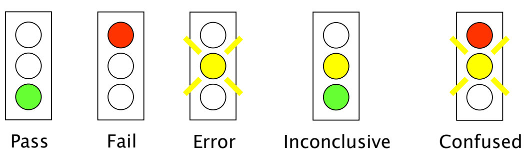
Verdicts



Verdicts explained

Verdict	*Explanation*
Pass	System under test has behaved correctly
Fail	System under test has behaved incorrectly
Error	Tester has behaved incorrectly
Inconclusive	Pass, but a certain test purpose has not been met
Confused	Fail-and-Error, result produces by an ambiguous specification (a special corner case)

Verdicts as traffic lights



Calculating verdict

- Verdict is calculated from a trace T and a specification S

verdict(T,S) $\in$ { pass, fail, error, conf }

- No inconc, because requires a test purpose

Pass verdict

verdict(T,S) = pass

if and only if

T $\in$ Tr(S)

Other verdicts

- Hence, T $\notin$ Tr(S) implies

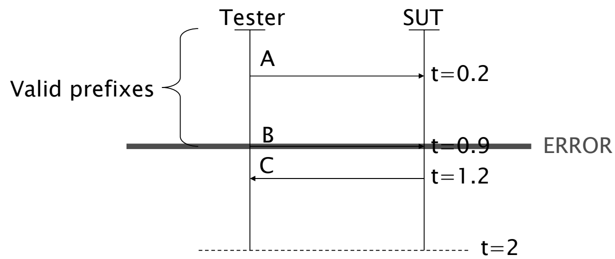
  verdict(T,S) $\in$ { fail, error, conf }

- There is one verdict for T $\in$ Tr(S), and three for the other case

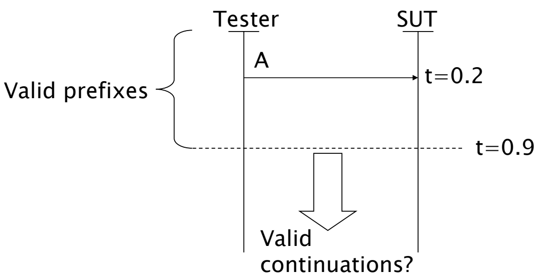
Other verdicts

- The problem: how to classify the cases T ∉ Tr(S) into
  - errors of the SUT (→ fail),
  - errors of the tester (→ error),
  - and those cases where the erring party cannot be defined (→ confused)?

Solution sketch (1)



Solution sketch (2)



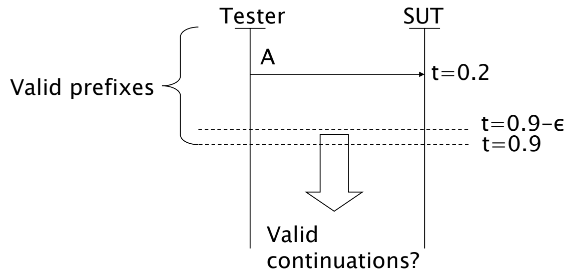
Solution sketch (3)

- All valid continuations differ from T first at input events?
  → error
- All valid continuations differ from T first at output events?
  → fail
- Otherwise
  → confused

Technicality

- The set of end time stamps for the valid prefixes of T can be either open or closed at the upper boundary
- Open set requires basically a limit construct (as usual)

Solution sketch (4)



Details

- Assume $T \notin Tr(S)$
- Let $V = Tr(S) \cap Pfx(T)$
  - Note: $\epsilon \in V$
- Let $K = \{ t \mid \exists E: <E, t> \in V \}$
- K is either
  - closed: [0, t], or
  - open: [0, t).

Example (closed set)

- $Tr(S) =$

$\cup \{ Pfx(<\{<A,t'>\},t>) \mid t \in [2,\infty), t' \leq 1 \}$

- $T = <\varnothing, 10>$
- Note that $T \notin Tr(S)$
- $V = \{ <\varnothing, t> \mid t \leq 1 \}$
- $K = [0, 1]$
- Especially $<\varnothing, 1>$ is in V, because $<\{<A,1>, 1.1>$ is valid

Example (open set)

- $Tr(S) =$

$\cup \{ Pfx(<\{<A,t'>\},t>) \mid t \in [2,\infty), t' < 1 \}$

- $T = <\varnothing, 10>$
- Note that $T \notin Tr(S)$
- $V = \{ <\varnothing, t> \mid t < 1 \}$
- $K = [0, 1)$
- Especially $<\varnothing, 1>$ is not in V, because for any $t' < 1$, event $<A,t'>$ should belong to the event set at time 1

Details continued

- Choose $\delta \in K$ (note: $0 \in K$ always, so K is not empty)
- Let X_δ denote the set of all valid extensions of $T[\delta]$ beyond the end time stamp of T

Details continued

- For every T' in X_δ, T' differs from T and $\Delta(T,T')$ is defined
- For every T', denote by α $T|_{\Delta(T,T')}$ if not τ
  - Otherwise denote by α $T'|_{\Delta(T,T')}$
  - Note: α can not be τ
- Let D_δ be the union of all α
- D_δ lists those events on which valid extensions of $T[\delta]$ differ from T

Details continued

- Assume there exists $\delta \in K$ such that $D_\delta \subseteq \Sigma_{in}$
  - Tester failure → error
- Assume there exists $\delta \in K$ such that $D_\delta \subseteq \Sigma_{out}$
  - SUT failure → fail
- Otherwise
  - undefined → confused

Details continued

- If K is closed, we can always choose $\delta = \max K$
- If K is open, we must choose a δ "close enough" the upper bound of K
  - $(\sup K) - \epsilon$ for $\epsilon > 0$

Disjointness

- $D_\delta \subseteq \Sigma_{in}$ and $D_\delta \subseteq \Sigma_{out}$ are disjoint conditions, because
  - $\delta \leq \epsilon$ implies $D_\epsilon \subseteq D_\delta$
  - D_δ is always non-empty
  - Σ_{in} and Σ_{out} are disjoint

Summary

- Is $T \in Tr(S)$?
  - Verdict is "pass"
- Else
  - Does there exists $\delta \in K$ such that $D_\delta \subseteq \Sigma_{out}$?
    - Verdict is "fail"
  - Otherwise, does there exists $\delta \in K$ such that $D_\delta \subseteq \Sigma_{in}$?
    - Verdict is "error"
  - Otherwise verdict is "confused"

Testing of Concurrent Systems 2004

Lecture 7

12th Oct 2004

Course this far

14.9	1	▸ Introduction ▸ General concepts ▸ Traces
	2	▸ Concurrent Scheme
21.9	3	▸ Traces, specifications
	4	▸ Seriality, execution introduction
28.9	5	▸ Test steps and execution
	6	▸ Test verdicts

"Scheme in everything"

- Until now, testing strategies and specifications have not had a structure
- We now consider Scheme programs as
  - implementations,
  - testing strategies,
  - testers, and
  - specifications.

Structure of a complete program

```
;; Definitions
(define …)
(define …)
(define …)
…
;; Entry point
(expr …)
```


Programs that generate behaviour

- Programs denote computational processes
- A computational process is characterized by its external behaviour, i.e. traces
- But we already have a function for generating traces, namely ξ

Behaviour through ξ

- Suppose i is a Scheme program
- One way to characterize the behaviour of the program i is the set

  $ETr(i) = \{ T \mid \exists s \in \mathbb{T} : P[i,s,T] > 0 \}$
- This is the set of all traces that are produced by *some* testing strategy with a non-zero probability
- Here we assumed that ξ was defined for Scheme programs

Defining ξ

- We assume a definition of ξ with the following properties:
  - Invalid input causes output ERR and program termination
    - ERR is a special symbol we reserve for this purpose
  - Division by zero or other run-time error causes output ERR and program termination
  - Termination of last thread → program termination
  - A terminated program does not produce any output whatsoever
  - Otherwise, assumed semantics of Scheme are preserved

Testing strategies (general)

- We assume that the set of testing strategies $\mathbb{T}$ contains at least all fixed input message sequences (a reasonable assumption).
- Hence, ETr(i) is *the* set of traces that a program can produce “against a suitable environment”.

Example

- Consider this program p:

```
(sync)
```

- ETr(p) contains traces void of events, and every trace that contains ERR and one or more input messages

Another example

- Consider this program q:

```
(define (run)
  (sync (input x (run))))
(run)
```

- ETr(q) contains all traces with only input messages; no ERR output is possible.

Summary of ETr(p)

- Set of traces that program p can generate with non-zero probability against at least one environment = testing strategy
- Invalid input or invalid computation causes program to halt → ERR, then no output

Programs as testing strategies

- Programs function as testing strategies “as implementations”
- We do not consider strategies that can crash
- We do not give more rigorous definition (at least now)
- Semantics implemented by ξ (as usual)

Programs as full testers

- Assume the existence of verdict-announcing functions:
  - (pass)
  - (fail)
- That’s it!

Example

- Tester for Echo Program:

```
(begin
  (sync (output “hello”
          (sync (input x
                   (if (equal? x “hello”)
                       (pass)))
                (wait 1 #f))))
  (fail))
```


Programs as specifications

- We have seen how a program can be used as an implementation
- We now turn to consider how a program can be used as a specification

Properties of specifications

- Suppose S is any specification
- Then Tr(S) is
  - prefix-complete
  - serial
  - non-empty

Interpreting programs as specifications

- A 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
- Hence, what is Tr(p) for a program p?

What is Tr(p)

- Is Tr(p) = ETr(p)?
  - No
  - ETr(p) contains traces where the system has halted due to an execution error (output ERR has been produced)
  - Execution error could be caused by invalidly received input
  - A trace that contains invalid input cannot be included in the set of valid traces!

Fix 1

- Let VTr(p) be that subset of ETr(p) that does not include ERR outputs:

  $VTr(p) = \{ \langle E,t \rangle \mid \langle E,t \rangle \in ETr(p) \wedge ERR \notin E \}$
- Could we postulate TR(p) = VTr(p)?
- No!
- VTr(p) is prefix-complete but not necessarily serial!

Non-serial program

- Program p below has non-serial set VTr(p):

```
(define (bug)
  (sync (wait 1 (/ 1 0))))
```

- All traces in Tr(p) below one second contain ERR, and thus are not in the set VTr(p)

Fix 2

- Let Tr(p) be the largest subset of VTr(p) that is serial
  - Being serial is a closure property
  - Hence this subset is well-defined
- Tr(p) is now serial, prefix-complete by construction
- If it is non-empty, then p is valid specification interpreted like this

Motivation

- No trace out of VTr(p) should be valid (largest subset of ETr(p) not containing ERRs)
- Suppose T belongs to VTr(p), but it has not arbitrarily long extensions
- Then an execution error is guaranteed after a finite time
- Hence T must be considered a phantom trace
- Tr(p) is now the largest subset of VTr(p) not containing these traces
- Tr(p) fulfils the properties required from a set of valid traces

Testing of Concurrent Systems 2004

Lecture 8

12th Oct 2004

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
  - 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

The "require" procedure

- Assume the following definition:

```
(define (require b)
  (if (not b) (/ 1 0)))
```


Require example

```
(define (echo)
  (sync (input x (require (integer? x))
             (wait-and-send x))))
(define (wait-and-send x)
  (sync (input y (wait-and-send x))
          (wait (/ (+ 1 (random 99)) 1000)
              (sync (output x (echo))))))
(echo)
```

Require example continued

- Let p be the program from previous slide
- Let T = <{"hello"$_{in}$, 1>}, 3>
- Note: T ∉ Tr(p)
- What is verdict(T,p)?

Verdict

- Verdict is ERROR
- The set of valid prefixes of T is
  { <∅,t> | 0 ≤ t ≤ 1}.
- Namely, "hello" causes the call to require to cause division by zero
  - Execution causes eventually ERR output
  - Hence the trace does not belong to the maximal serial subset
  - The input "hello" is the problem → tester error

Use of require

- Require is a device for "intensional" specifications
- Can be mind-boggling
- Consider the following example

Require trick

```
(define (guess)
 (let ((v (any-integer)))
  (sync (input x
                (require (= v x))
                (sync (wait 0.1
                         (sync (output "ok"
                                   (wait-for-ever)))))))
(define (wait-for-ever)
   (sync (input x (wait-for-ever))))
(guess)
```


Require trick (2)

- Let T = <{<3, 0>}, 10>
- What should be verdict(T, p)???
- The solution is…

Require trick conclusion

- … FAIL.
- Can you understand why?

Testing of Concurrent Systems 2004

Lecture 9
19th Oct 2004

Course this far

14.9	1	▸ Introduction, general concepts, traces
	2	▸ Concurrent Scheme
21.9	3	▸ Traces, specifications
	4	▸ Seriality, execution introduction
28.9	5	▸ Test steps and execution
	6	▸ Test verdicts
12.10	7	▸ Scheme programs as implementations, testing strategies, testers, and specifications
	8	

Summary of last lecture

- Scheme programs as implementations and specifications
- ETr(p), VTr(p), Tr(p)

Conformance?

- What does it mean that a system conforms to a specification?
  - System functions as specified
  - System passes all tests
  - Which “all” tests?
  - System passes every “test” that is “correct”
  - What is “a test”? What is “a correct test?”

What is “a test”?

- A test = ?
  - a specific <u>testing strategy</u>
  - a specific <u>test execution trace</u>
  - a specific <u>tester</u>
  - a specific <u>tester execution</u>
- A correct test = ?
  - A test execution trace with verdict ≠ ERROR
  - A testing strategy or tester that “never works illegally”
    - What does this mean?

Correct testing strategies

- A testing strategy s is <u>correct with respect to a specification</u> S if for <u>any</u> implementation i:

  $P[i,s,T] > 0 \Rightarrow \text{verdict}(T, S) \neq \text{ERROR}$
- Denote by CT(S) the set of all correct testing strategies with respect to S

Synthetic correct strategies

- A correct testing strategy can be (informally) constructed by the following loop:
  - Guess the next action (send/wait) so that a valid trace extension will result
  - Execute the chosen action, observing the actions of the SUT
  - Restart loop
- More on this on the second half!
- Shows that correct testing strategies exist
  - Possible because of the seriality of valid set of traces
- In real life computationally intensive

Correct strategies ctd

- If we assume these synthetic strategies belong to the set of available testing strategies…
- … then all correct and failing behaviours can be constructed against correct testing strategies.
- Make this assumption for now.

Trace taxonomy

- Let p, p' be Scheme programs
- For every trace T, one of the following is true:
  - There exists s ∈ CT(p') such that
    P[p, s, T] > 0
  - There exists s, but none in CT(p'), such that
    P[p, s, T] > 0
  - There does not exist any s such that
    P[p, s, T] > 0

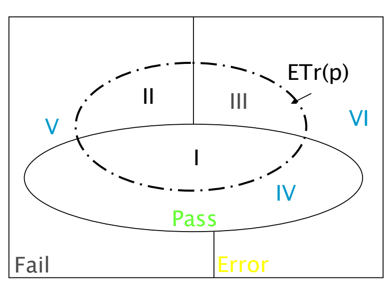
Trace taxonomy ctd

- Furthermore, for every trace T it holds that verdict(T, p') is one of PASS, FAIL, ERROR
- We ignore ambiguous specifications (→ verdict CONFUSED) for now

Matrix for a trace T

Verdict → Condition	PASS	FAIL	ERROR
∃s ∈ CT(p'): P[p, s, T] > 0	Possible	Possible	Not possible (def. correct strategy)
∃s: P[p, s, T] > 0 ∀s: P[p, s, T] > 0 ⇒ s ∉ CT(p')	Not possible (synthetic testers)	Not possible (synthetic testers)	Possible
∄s: P[p, s, T] > 0	Possible	Possible	Possible

Trace map of a general system



CORRECT AND INCORRECT TESTERS

I: correct, producible behaviour

II: incorrect, producible

INCORRECT TESTERS ONLY

III: producible behaviour against malfunctioning environment only

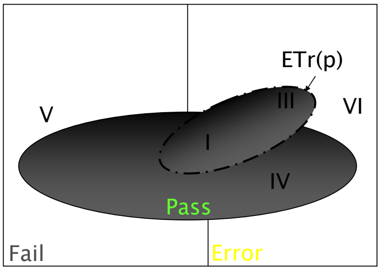
NO TESTERS AT ALL

IV: correct behaviour not implemented

V: incorrect behaviour not implemented

VI: behaviour involving malfunctioning environment that has not been implemented

Trace map of a correct system



CORRECT AND INCORRECT TESTERS

I: correct, producible behaviour

INCORRECT TESTERS ONLY

III: producible behaviour against malfunctioning environment

NO TESTERS AT ALL

IV: correct behaviour not implemented

V: incorrect behaviour not implemented

VI: behaviour involving malfunctioning environment that has not been implemented

Correct system ctd.

- If we restrict ourselves to correct testers, then
- all behaviour that can be generated is included within the set of valid traces Tr(S).

Execution against correct strategies

- Recall

  $ETr(i) = \{ T \mid \exists s \in \mathbb{T} : P[i,s,T] > 0 \}$
- Define now

  $ETr(i, S) = \{ T \mid \exists s \in CT(S) : P[i,s,T] > 0 \}$
- Here CT(S) is the set of testing strategies correct with respect to S
- Note $ETr(i, S) \subseteq ETr(i)$ for all S

Continued…

- We have now eliminated the ERROR verdict
- Suppose for all $s \in CT(p')$,

  $P[p, s, T] > 0$ implies $verdict(T, p') \neq$ FAIL
- Then (assuming unambiguous specifications),

  $P[p, s, T] > 0$ implies $verdict(T, p') =$ PASS
- Hence, $\underline{ETr(p, p') \subseteq Tr(p')}$

Conclusion

- We have thus reduced the conformance of a program p to a specification p' to the equation

  $\underline{ETr(p, p') \subseteq Tr(p')}$
- This is the underlying notion of conformance in the known theory of formal conformance testing

Conclusion ctd.

- Conformance = trace inclusion
  - Traces generated by implementation are included in those generated by specification
  - Incorrectly generated/out-of-specification traces must be excluded
  - Note: no explicit mention of single testing strategies above!

Implications

- Quantifying over all testers leads to simple trace set inclusion
- This trace set inclusion can be also checked for <u>directly</u> under suitable conditions → model checking
- Thus formal conformance testing = "partial model checking"

Note

- Note that ETr(p, p) is not necessarily a subset of Tr(p)
- There are systems that are not conforming to themselves!

Example

```
(let ((x (choose)))
  (if x
      (wait-for-ever)
      (begin
         (sync (output "alert"))
          (/ 1 0)
         (wait-for-ever))))
```


Example ctd.

- This program p is not conforming to itself: ETr(p, p) is not a subset of Tr(p)
- Can you see why?
- Systems with internal computation errors are not self-conforming
- What about invalid inputs?

Example 2

```
(sync)
```

- This program p is self-conforming
- Every testing strategy in CT(p) is silent
- Hence, the resulting traces are completely void of events
- Hence ETr(p, p) contains only silent traces, which are contained in Tr(p)

Example 3

```
(let ((x (receive-a-truth-value)))
  (if x
      (wait-for-ever)
      (begin
         (sync (output "alert"))
         (/ 1 0)
         (wait-for-ever))))

(define (receive-a-truth-value)
   (sync (input x (require (boolean? x)) x)))
```

Example ctd.

- This program p <u>is</u> conforming to itself
- The input #f (false) is not valid, because it leads unavoidably to execution error; hence traces with #f input are not in Tr(p)
- ETr(p, p) is a subset of Tr(p)

Testing of Concurrent Systems 2004

Lecture 10

19th Oct 2004

Specification-based testing algorithms

- Algorithms for running testing, based on a specification

Basic on-the-fly algorithm

$E := \emptyset$, clock := 0

while [true]

  $X_\tau := \{ \langle E, clock+\epsilon \rangle \mid \epsilon > 0, \langle E, clock+\epsilon \rangle \in Tr(S) \}$

  $X_{in} := \{ \langle E \cup \langle m, clock \rangle, clock+\epsilon \rangle \mid m \in \Sigma_{in}, \epsilon > 0,$
      $\langle E \cup \langle m, clock \rangle, clock+\epsilon \rangle \in Tr(S) \}$

  $X_{out} := \{ \langle E \cup \langle m, clock \rangle, clock+\epsilon \rangle \mid m \in \Sigma_{out}, \epsilon > 0,$
      $\langle E \cup \langle m, clock \rangle, clock+\epsilon \rangle \in Tr(S) \}$

  $N := X_\tau \cup X_{in} \cup X_{out}$

  if [$N = \emptyset$] then <u>FAIL</u>

  if [stopping criterion] then <u>PASS</u>

  choose $T = \langle E', t \rangle$ from N

  if $T|_{clock} \in \Sigma_{in}$ then { send $T|_{clock}$, $E := E \cup \langle T|_{clock}, clock \rangle$ }

  wait for input until t   // t > clock

  if [input m received at time t' (clock $\le$ t' < t)]

    then $E := E \cup \langle m, t' \rangle$, clock := t'

    else clock := t

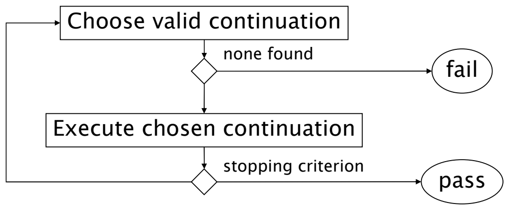
Correctness arguments

- <E, clock> is "current" trace
- If there is <u>no</u> proper extension of <E,clock> 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, immedately 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?

- Suppose the algorithm produces trace T such that verdict(T, S) = ERROR
- Hence $T \notin Tr(S)$. There exists time t at which T has deviated from the longest valid prefix of it
- Every proper extension of a prefix $T[t-\epsilon]$ for sufficiently small ϵ differs from T by change of input behaviour
- One of the prefixes of T is the trace at the last loop of the algorithm where the trace is still valid; the next trace differs from correct traces first by input behaviour
- But input behaviour is always chosen so that it does not lead to outside valid behaviour (Tr(S))
- Hence ERROR verdict is impossible
- This is a correct tester for S, regardless of the choice structure

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, no matter what the SUT does
- Single-agent planning = co-operative planning = plan that assumes that SUT co-operates
- Adversarial planning = planning against enemy = plan that assumes that SUT actively resists testing
- 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
- Conformant plan: none, as there is no way to enforce the die to give 6
- Single-agent plan: roll once—the die will co-operate and give 6
- Adversarial plan: 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

Summary

- Basic on-the-fly algorithm
- Planning types
- Stopping criterion

Testing of Concurrent Systems 2004

Lecture 11

16th Oct 2004

Course this far

14.9	1	Introduction, general concepts, traces
	2	Concurrent Scheme
21.9	3	Traces, specifications
	4	Seriality, execution introduction
28.9	5	Test steps and execution
	6	Test verdicts
12.10	7	Scheme programs as implementations, testing strategies, testers, and specifications
	8	
19.10	9	Conformance = trace inclusion
	10	Basic on-the-fly testing algorithm

Economics of testing

- We know <u>what</u> testing is
- We know <u>how</u> we can test (at least basically)
- But <u>why</u> we should test?

Testing is economic activity

- Testing costs
  - Money
  - Working time
  - Other resources
- Because it costs, there must be a pay-off
- What is the pay-off from testing?

Pay-off of testing

- Detection of bugs or faults
  - Only a known bug can be fixed
  - Knowledge of a bug is valuable
- Increased confidence
  - = reduced risk of malfunctioning
  - Can be obtained <u>without changing</u> the SUT!

Rationale for testing

- We pay the cost of testing in order to reduce the risk of system malfunctioning
  - Additionally, we can spot defects, but we do not know beforehand if that will happen
- How risk reduction happens?
- How useful is it? How can we quantify it?
- We are comparing two basically incompatible things: money and risk

Basic utility theory

- Assume you make a choice between a set of alternatives $\alpha_1, \alpha_2, \ldots$
- A <u>rational</u> choice is to choose the alternative that is <u>most useful</u> = <u>has highest utility</u>

Utility values

- Denote the utility of an alternative α by $u(\alpha)$; it is a real number
- Rational agent chooses alternative α_i such that $u(\alpha_i)$ is the maximum of all utilities (assume the maximum exists)

Lotteries

- A lottery L is a probability distribution over a set of alternatives A:

$$L : A \rightarrow [0,1]$$

- The expected utility theory assumes that the utility of L is given by

$$u(L) = \Sigma_{\alpha \in A} L(\alpha) u(\alpha)$$

Utility of money

- Money has in general nonlinear utility
- Compare receiving 1,000,000 € with taking part in the lottery L over 0 €, 2,000,000 € such that L(0 €) = ½ and L(2,000,000 €) = ½. Which one would you choose?
- We assume linear utility, so we can use money as unit of utility (for the sake of concreteness)

Failure models

- Recall that a failure model is a function from specifications to probability distributions over implementations

$$\mu : \mathbb{S} \to (\mathbb{I} \to [0, 1])$$

Transforming distributions

- Assume S is a specification and $\psi = \mu(S)$
- Unknown SUT i is chosen according to μ(S)—the a priori distribution
- We test i with strategy s, observing trace T (with verdict PASS). What is the a posteriori distribution of SUTs?

Linking distributions

- Denote the a posteriori distribution by ψ'
- ψ is transformed to ψ' by s, T
- How?
- We will employ the <u>**Bayes' Rule**</u>

Bayes' Rule

- The Bayes' Rule is a basic rule of conditional probability:

$$P(B|A) = P(A|B)\, P(B) \;/\; P(A)$$

- Derivation:
  - P(B|A) =
  - P(A,B)/P(A) =
  - P(A,B) P(B) / [P(B) P(A)] =
  - P(A|B) P(B) / P(A)

Compute ψ'

- We have produced trace T against unknown implementation with strategy s, a priori distribution of implementations being ψ
- The a priori probability for trace T is

$$\Sigma_i\, \psi[i]\, P[i,s,T]$$

- The a priori probability for implementation i is ψ[i]
- What is the a posteriori probability distribution for an implementation i?

Compute ψ' (2)

- $\psi'(\psi,s,T)[i] =$

  $P[i,s,T]\ \psi[i] \big/ \left(\Sigma_{i^*}\ \psi(i^*)\ P[i^*,s,T]\right)$
- If $P[i,s,T] = 0$ then $\psi'[i] = 0$
- If $\psi[i] = 0$ then $\psi'[i] = 0$
- if $P[i,s,T] = kP[i',s,T]$, then $\psi'[i]/\psi[i] = k\psi'[i']/\psi[i']$

Correct implementations

- Let A(S) be the set of (absolutely) correct implementations of specification S
- The a priori probability for a correct system is $C = \Sigma_{i\in A(S)}\psi[i]$
- The a posteriori probability for a correct system after testing is $C' = \Sigma_{i\in A(S)}\psi'(\psi,s,T)[i]$
- If testing results in PASS, is it automatically true that $C' \geq C$..?

Increasing correctness

- The answer is, unfortunately, no.
- It is possible that $C' < C$.
- This is a strange paradox of testing: sometimes a specific test run yields verdict PASS, but still decreases the probability of having a correct system.
- But we don't dwell longer on this.

Economic implications

- Suppose an correct system has utility of X (€), incorrect system has utility 0
- The expected utility of unknown SUT is CX before testing and C'X after testing
- If $C' > C$, it pays off to pay less than $(C' - C)X$ for testing

Shortcomings

- In general we can't know C' before we have done the testing
- Also, the utility of a correct system or the disutility of an incorrect one is not constant. It depends on use, and is itself probabilistic.
- More on this on the next lecture…

Testing of Concurrent Systems 2004

Lecture 12

26th Oct 2004

Incorrect systems introduction

- Denote by $P_n[i,s,T]$ the probability that trace T is produced against implementation i with strategy s in exactly n test steps.
- The probability that the execution after n steps is incorrect w.r.t. S is given by

$$F_n[i,s,S] = \Sigma_{T \notin Tr(S)} P_n[i,s,T]$$

Properties of F_n

- $F_0[i,s,S] = 0$
- $F_n[i,s,S] \leq F_{n+1}[i,s,S]$

Expected length of correct execution

- Suppose the series

$$E = \Sigma_{i>0} (i - 1)(F_i - F_{i-1})$$

converges. Then E is the expected length of a correct execution before failure.
- $F_i - F_{i-1}$ is the probability that the first failure occurs at i^{th} step.

Alternative form

- E can be computed also as

$$\Sigma_{i>0} (1 - F_i)$$

- Summing up the previous series upto k, we get

$$(\Sigma_{0<i\leq k} -F_i) + kF_{k+1}$$

- Because the series converges, lim $n\rightarrow\infty$ $F_n = 1$. Hence in the limit, we get

$$(\Sigma_{0<i\leq k} -F_i) + k = (\Sigma_{0<i\leq k} 1-F_i).$$

When E does not converge

- What if E is not converging?
- If lim $n\rightarrow\infty$ $F_n = 0$, the system is absolutely correct with respect to the strategy s
- Otherwise (larger limit), the relative probability of system failure must decrease in time
  - If $(1 - F_n)/(1 - F_{n-1}) < \alpha < 1$ (α constant), the series converges

E does not converge (ctd)

- Those cases where E does not converge but the system is not absolutely correct can be ruled out as “unnatural”
  - E.g. systems that are capable of failing only at “system start” but not later—what is a system start? Is reset not allowed never afterwards?
- This leaves us with two system classes w.r.t. a strategy s
  - Absolutely correct systems
  - Those with finite expected correct execution length

Economic considerations

- Assume every use step yields benefit B, and every system failure costs F
- The discounted benefit per use step is B – F/E
- If B – F/E < 0, system is useless
- For absolutely correct system (with respect to a strategy), the discounted benefit is just B

Use strategy

- If we fix a use strategy s, we can compute the expected benefit from a distribution ψ as usual:

$$\Sigma_i\, \psi[i](B - F/E_i)$$

- After testing , this becomes

$$\Sigma_i\, \psi'[i](B - F/E_i)$$

- But we can't still quantify the benefit of testing...

… ctd …

- … because we do not know how <u>long</u> the system is used
- Assume life cycle of N steps
- Total increase of utility is

$$N\, \Sigma_i\, (\psi'[i] - \psi[i])(B - F/E_i)$$

- Bounded above by NB
- Computing in practice basically impossible—why?

Notes

- Analysis of this kind depends heavily on the failure model μ
- How it is obtained?
- Can the calculations be carried actually out?
- What is the utility?

Testing of Concurrent Systems 2004

Lecture 13
2nd Nov 2004

Course this far

14.9	1	Introduction, general concepts, traces
	2	Concurrent Scheme
21.9	3	Traces, specifications
	4	Seriality, execution introduction
28.9	5	Test steps and execution
	6	Test verdicts
12.10	7	Scheme programs as implementations, testing strategies, testers, and specifications
	8	
19.10	9	Conformance = trace inclusion
	10	Basic on-the-fly testing algorithm
26.10	11	Rational decision, "economics of testing"
	12	

Review

- Comments from the last lecture:
  —"What is the use of this?"
- After the infamous economics of testing lecture, we move (back) to something more concrete, namely
- testing algorithms.

Basic on-the-fly algorithm

- We review the algorithm from lecture 10 and insert a small fix
- The original algorithm is correct, but this fix makes later developments more straightforward

Basic on-the-fly algorithm

```
E := ∅, clock := 0
while [ true ]
    X_T := { <E, clock+ϵ> | ϵ > 0, <E, clock+ϵ> ∈ Tr(S) }
    X_in := { <E ∪ <m, clock>, clock+ϵ> | m ∈ Σ_in, ϵ > 0,
              <E ∪ <m, clock>, clock+ϵ> ∈ Tr(S) }
    X_out := { <E ∪ <m, clock>, clock+ϵ> | m ∈ Σ_out, ϵ > 0,
              <E ∪ <m, clock>, clock+ϵ> ∈ Tr(S) }
    N := X_T ∪ X_in ∪ X_out
    if [ N = ∅ ] then FAIL
    if [ stopping criterion ] then PASS
    choose T = <E', t> from N
    if T|clock ∈ Σ_in then { send T|clock, E := E ∪ < T|clock, clock> }
    wait for input until t                  // t > clock
    if [ input m received at time t' (clock ≤ t' < t) ]
        then E := E ∪ <m, t'>, clock := t' + ϵ for a "very" small ϵ
        else clock := t
```

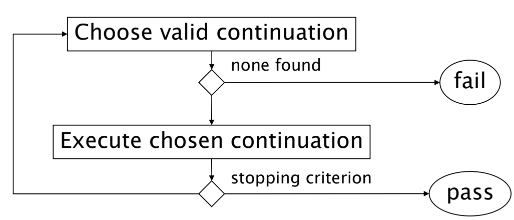

Purpose of the fix

- Now <E, clock> is always a valid trace object
- This did not hold previously, even though the algorithm was correct
- A practical implementation can handle the issue differently

Review continues

- We review the abstract version

Abstract version



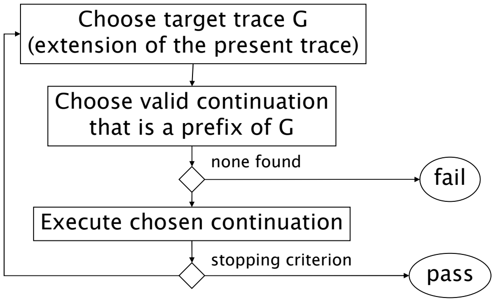
A planning version

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

Plan-oriented testing algorithm

```
E := ∅, clock := 0                                          New choice
while [ true ]
    Choose a suitable G from Tr(S) s.t. <E,clock> ≺ G
    X_τ := { <E, clock+ϵ> | ϵ > 0, <E, clock+ϵ> ≼ G }
    X_in := { <E ∪ <m, clock>, clock+ϵ> | m ∈ Σ_in, ϵ > 0,      Reduced
              <E ∪ <m, clock>, clock+ϵ> ≼ G }                   options
    X_out := { <E ∪ <m, clock>, clock+ϵ> | m ∈ Σ_out, ϵ > 0,
              <E ∪ <m, clock>, clock+ϵ> ≼ G }
    N := X_τ ∪ X_in ∪ X_out
    if [ N = ∅ ] then FAIL
    if [ stopping criterion ] then PASS
    choose T = <E', t> from N
    if T|_clock ∈ Σ_in then { send T|_clock, E := E ∪ < T|_clock, clock> }
    wait for input until t                 // t > clock
    if [ input m received at time t' (clock ≤ t' < t) ]
          then E := E ∪ <m, t'>, clock := t' + ϵ for a "very" small ϵ
          else clock := t
```


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 UniversalPropertyCheck 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
  - We will see a concrete implementation

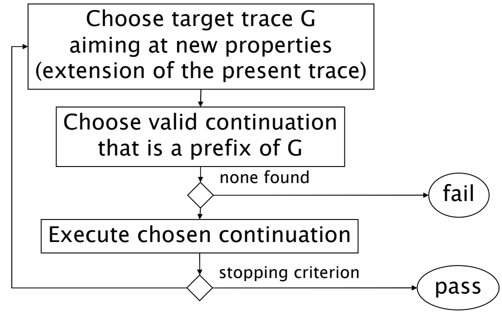
Property covering (ctd.)

- Furthermore, assume there exists another procedure PlanForMoreProperties 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

Property-covering testing algorithm

```
E := ∅, clock := 0, P := ∅
while [ true ]
    P := P ∪ UniversalPropertyCheck(<E, clock>, S)
    G := PlanForMoreProperties(P, <E, clock>, S)
    if [ no G found ]
          Choose a suitable G from Tr(S) s.t. <E,clock> ≺ G
    X_τ := { <E, clock+ϵ> | ϵ > 0, <E, clock+ϵ> ≼ G }
    X_in := { <E ∪ <m, clock>, clock+ϵ> | m ∈ Σ_in, ϵ > 0,
             <E ∪ <m, clock>, clock+ϵ> ≼ G }
    X_out := { <E ∪ <m, clock>, clock+ϵ> | m ∈ Σ_out, ϵ > 0,
             <E ∪ <m, clock>, clock+ϵ> ≼ G }
    N := X_τ ∪ X_in ∪ X_out
    if [ N = ∅ ] then FAIL
    if [ stopping criterion ] then PASS
    choose T = <E', t> from N
    if T|_clock ∈ Σ_in then { send T|_clock, E := E ∪ < T|_clock, clock> }
    wait for input until t                  // t > clock
    if [ input m received at time t' (clock ≤ t' < t) ]
          then E := E ∪ <m, t'>, clock := t' + ϵ for a "very" small ϵ
          else clock := t
```


Abstract version



Testing of Concurrent Systems 2004

Lecture 13

2nd Nov 2004

A dive deeper

- How do we check if T ϵ Tr(S)?
- How do we compute the "properties" that a trace "necessarily" covers?
- How do we compute goal traces?

State space based computation

- The execution function ξ gives <u>external behaviour</u>, but it thus abstracts away the "internals" of a specification
- 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' ϵ next(s), we write also s→s'

State spaces

- Assume we can associate with a specification
  - an initial state $s_0=\langle c_0,\langle\emptyset,0\rangle\rangle$
  - next state relation
- $Tr(S) = \{ T \mid \exists\langle c,T\rangle : s_0 \to^* \langle c,T\rangle \}$
- We assume that the seriality requirement is fulfilled implicitly in the state space
  - But this is not necessarily the case in reality

Basic trace inclusion check algorithm

```
W := {s0}
V := ∅
While W≠∅
   Choose <c,T> from W
   If T = T*
        Return FOUND
   Else if T ≺ T*
        W := W – {<c,T>}
        V := V ∪ {<c,T>}
        W := W ∪ (next(<c,T>) – V)
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 $s \to_P s'$ if there is a transition from s to s' with properties P

UniversalPropertyCheck(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*
        W := W – {<<c,T>,π>}
        V := V ∪ {<<c,T>, π>}
        N := { <s', π'> | s →Q s', π' = π ∪ Q }
        W := W ∪ (N – V)
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

PlanForMoreProperties(P,T*,S)

```
W := {<s0, ∅>}
V := ∅
While W≠∅
    Choose <<c,T>,π> from W
    If T ≼ T* or T* ≼ T
          If π ⊈ P and T* ≺ T
                    If (UniversalPropertyCheck(T,S) ⊈ P)
                              Return T
          Else
                    W := W – {<<c,T>,π>}
                    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 UniversalPropertyCheck, it holds that at least one way to reach the trace T implies new properties
- The UniversalPropertyCheck call is used to ensure that this holds for all alternative executions as well

Copyright © Antti Huima 2004. All Rights Reserved.

Discussion

- Property = interesting feature in specification
- For example, a property = a state in a state chart model, or a Scheme expression in a Scheme reference implementation
- Intuition: it is good to exercise "many parts" of reference implementation rather than "few parts"
- But…

Copyright © Antti Huima 2004. All Rights Reserved.

Discussion (ctd)

- … as mentioned on the "economics" lecture, it is impossible to <u>prove</u> that this would be a good thing
- So just a heuristic

Copyright © Antti Huima 2004. All Rights Reserved.

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)
  - …

Copyright © Antti Huima 2004. All Rights Reserved.

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. All Rights Reserved.

Symbolic execution

- If next(s) sets are infinite, the 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[s_0]}
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

Testing of Concurrent Systems 2004

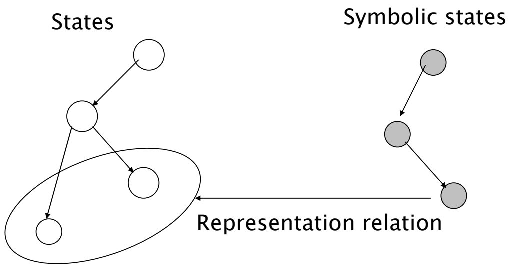
Lecture 15

15th Nov 2004

Course this far

14.9	1	Introduction, general concepts, traces
	2	Concurrent Scheme
21.9	3	Traces, specifications
	4	Seriality, execution introduction
28.9	5	Test steps and execution
	6	Test verdicts
12.10	7	Scheme programs as implementations, testing strategies, testers, and specifications
	8	
19.10	9	Conformance = trace inclusion
	10	Basic on-the-fly testing algorithm
26.10	11	Rational decision, "economics of testing"
	12	
2.11	13	Testing algorithms, planning, property covering
	14	State-space based algorithms

Symbolic states

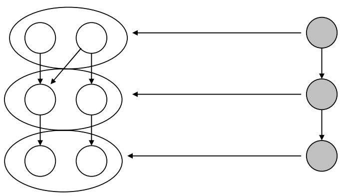


Representation

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

Axiom

- If $z \rightarrow z'$, then
  $\gamma(z') = \{ s' \mid \exists s \in \gamma(z): s \rightarrow s' \}$



Operations for symbolic states

- Emptiness check

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

- Intersection

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

- Subsumption relation

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

Symbolic successors

- $\mathrm{Next}(z) = \{ z' \mid z \rightarrow z' \}$
- Axiom 2:

  $$s \in \gamma(z),\ s \rightarrow s' \text{ implies}$$

  $$\exists z' \in \mathrm{Next}(z) : s' \in \gamma(z')$$

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]}   W = {{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

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

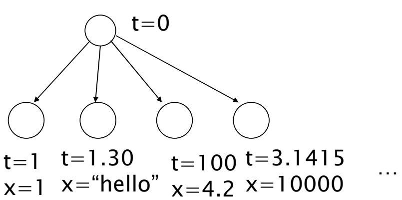
Compute successors but filter out states whose traces are not prefixes of T*

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)

Symbolic states: example

(sync (input x
  (sync (wait 0.1
    (sync (output (+ x 1) (halt))))))



?

Discussion

- The symbolic state space depicts the whole infinite state space, but is in the example finite
- Individual states are represented <u>symbolically</u> as individual solutions to a constraint set

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!

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

More details

- 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

- $c = [t \to X1, x \to X2, \ldots]$
- $T = \langle\{\langle X2_{in}, X1\rangle, \langle X4_{out}, X3\rangle\}, X3\rangle$
- $C = \{X1>0, X3=X1+0.1, \text{number } X2, X4=X2+1\}$
- <<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=<<c, T>, C> and z'=<<c', T'>, C'> 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

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 solvability 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 ⇒ C', check for the solvability of C ∧¬C'
- Assumes now that constraint sets are closed also under negation → full Boolean closure

More algorithms

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

Testing of Concurrent Systems 2004

Lecture 16

15th Nov 2004

Symbolic execution of Scheme

- Let's have a simplified look on the stack-based execution of Scheme

Scheme execution

Value stack

Term stack (let ((x (+ 1 2))) (sync (output x)))

Environment

Scheme execution

Value stack

Term stack (+ 1 2) (BIND x) (sync (output x)) (RESTORE)

Environment

Scheme execution

Value stack

Term stack | + 1 2 (APP 2) (BIND x) (sync (output x)) (RES

Environment

Scheme execution

Value stack | primitive +

Term stack | 1 2 (APP 2) (BIND x) (sync (output x)) (RESTO

Environment

Scheme execution

Value stack | integer 1 | primitive +

Term stack | 2 (APP 2) (BIND x) (sync (output x)) (RESTOR

Environment

Scheme execution

Value stack | integer 2 | integer 1 | primitive +

Term stack | (APP 2) (BIND x) (sync (output x)) (RESTORE)

Environment

Scheme execution

Value stack | integer 3

Term stack | (BIND x) (sync (output x)) (RESTORE)

Environment

Scheme execution

Value stack

Term stack | (sync (output x)) (RESTORE)

Environment | x := integer 3

Scheme execution

Value stack $\boxed{\#f}$

Term stack $\boxed{\text{(RESTORE)}}$

Environment $\boxed{\text{x := integer 3}}$

Events <3, 0>

Scheme execution

Value stack $\boxed{\#f}$

Term stack

Environment

Copyright © Antti Huima 2004. All Rights Reserved.

How does this work symbolically?

- Straightforward Scheme executor manipulates concrete datums (integers, booleans, …)
- Symbolic Scheme executor manipulates <u>constraint variables</u> as datums

Copyright © Antti Huima 2004. All Rights Reserved.

Scheme execution

Value stack

Term stack $\boxed{\text{(sync (input x (if (> x 5) (gorblex x))))}}$

Environment

Copyright © Antti Huima 2004. All Rights Reserved.

Scheme execution

Value stack

Term stack $\boxed{\text{(if (> x 5) (gorblex x)) (RESTORE)}}$

Environment $\boxed{\text{x := X1}}$

Copyright © Antti Huima 2004. All Rights Reserved.

Scheme execution

Value stack

Term stack $\boxed{\text{(> x 5) (TEST (gorblex x) \#void) (RESTORE)}}$

Environment $\boxed{\text{x := X1}}$

Copyright © Antti Huima 2004. All Rights Reserved.

Scheme execution

Value stack

Term stack | > x 5 (APP 2) (TEST (gorblex x) #void) (REST

Environment | x := X1

Scheme execution

Value stack | Primitive >

Term stack | x 5 (APP 2) (TEST (gorblex x) #void) (RESTOI

Environment | x := X1

Scheme execution

Value stack | Variable X1 | Primitive >

Term stack | 5 (APP 2) (TEST (gorblex x) #void) (RESTORE

Environment | x := X1

Scheme execution

Value stack | integer 5 | variable X1 | primitive >

Term stack | (APP 2) (TEST (gorblex x) #void) (RESTORE)

Environment | x := X1

Scheme execution

Value stack | variable X2

Term stack | (TEST (gorblex x) #void) (RESTORE)

Environment | x := X1

Constraints $X2 \Leftrightarrow (X1 > 5)$, number X1

Scheme execution

Value stack

Term stack | (gorblex x) (RESTORE)

Environment | x := X1

Constraints $X2 \Leftrightarrow (X1 > 5)$, number X1, X2 = #t

Discussion

- Symbolic Scheme execution is a concrete instance of the symbolic state space exploration idea
- Can be used to implement formal conformance testing

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

Testing of Concurrent Systems 2004

Lecture 17

23rd Nov 2004

Course this far

14.9	1	Introduction, general concepts, traces
	2	Concurrent Scheme
21.9	3	Traces, specifications
	4	Seriality, execution introduction
28.9	5	Test steps and execution
	6	Test verdicts
12.10	7 8	Scheme programs as implementations, testing strategies, testers, and specifications
19.10	9	Conformance = trace inclusion
	10	Basic on-the-fly testing algorithm
26.10	11 12	Rational decision, "economics of testing"

2.11	13	Testing algorithms, planning, property covering
	14	State-space based algorithms
16.11	15	Symbolic state space exploration
	16	Symbolic execution of Scheme

Topics today

- The classic IOCO theory
- Critique of IOCO

Ioco theory

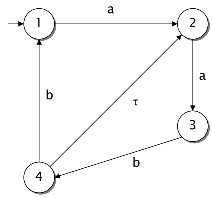
- The "classic theory"
- Often referred to as the "ioco" testing theory and is quite well known among the academic peoples
- A framework developed by Tretmans, Heerink et al.
- Dates to early 90's

Ioco 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 <u>labeled transition system</u> 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.

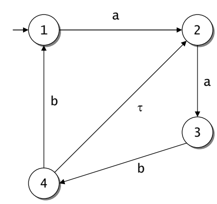


S={1,2,3,4}
L={a,b}
T={<1,a,2>,<2,a,3>,<3,b,4>,
<4,τ,2>,<4,b,1>}
S_0=1

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.

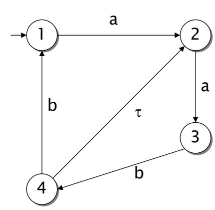
ε	a	aa
aab	aabb	aaba
aabba	aabab	aababa

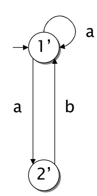


Parallel composition

- 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 <u>directional</u> but completely <u>symmetric</u>.
  - Can be therefore called a "handshake".

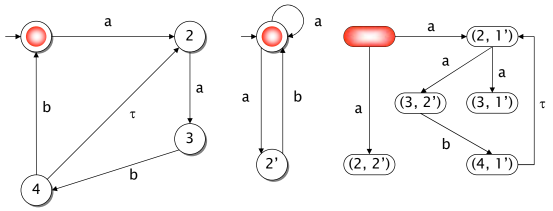
Example





- 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

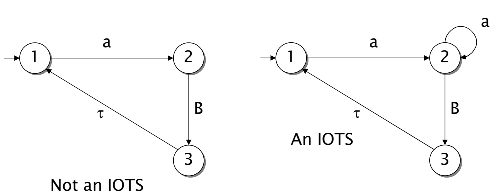
- In parallel composition, the two LTSs can take step with label a ($\neq \tau$) only if they do that together.
- This means that if a models, say, a message from Tester to SUT, then the SUT can refuse to receive the message (just by not having an outgoing transition with the label a).
- This is disturbing, 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}.



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 operational 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 t confirms i w.r.t. s if
  $\text{obs}(t \parallel i) \subseteq \text{obs}(t \parallel s)$.
  (All the three entities are IOTSs).
- We can now say that an implementation i conforms to a specification 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 can 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 can 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

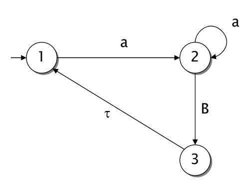
- An alternative way to define the same result is given next.
- i $\leq_{iot}$ s iff
  traces(i) $\subseteq$ traces(s) and Qtraces(i) $\subseteq$ Qtraces(s)
  where Qtraces(L) is the set of those traces of L after which L can be in a state where only transitions labeled by inputs are possible (i.e. L 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!

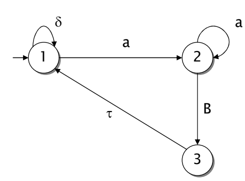
Quiescence…

- 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 can send out a meta-message δ which denotes "I'm waiting for input" or "I'm quiescent".





Repetitive Quiescence (ctd)

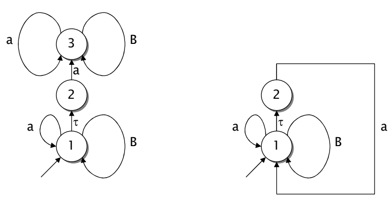
- The name for δ is "suspension".
- We call the traces of an IOTS with this extension (can produce δ when no output is possible) "suspension traces", denoted by Straces(*L*).

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.
- <u>This is the ioco testing relation.</u>

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 <u>in the midst</u> of a test run.



Testing of Concurrent Systems 2004

Lecture 18

23rd Nov 2004

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

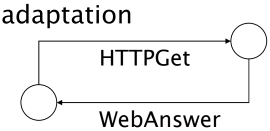
- How to create a small finite state machine (i.e. LTS) from a specification generating a big/infinite state space?
- Drop out details
- Replace data with abstract placeholders

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



Timing?

- The ioco theory has a weak notion of time: quiescence
- Quiescence corresponds to an abstract timeout
- However, there are no “quiescences of different length”
- Time is handled abstractly

Adding time

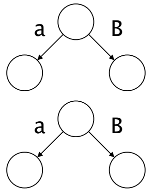
- We could extend the input and output alphabets to include time stamps (as the events in our general framework)
- Then, however, both tester and SUT LTSes must become infinitely large and acyclic
- In fact, we would have some form of an alternative representation of our trace sets

Adding time (ctd)

- But problems remain
  - Seriality
  - Progressivity
  - More definitions would be needed
- The notion of quiescence would become redundant
  - It is impossible to detect an “infinitely long” quiescence in a practical testing setup
- Tretmans et al have been working on a timed extension

Counter-intuitive synchronization

- Consider the tester and the SUT on the right (a is input, B output)
- How do you interpret this intuitively?
- How tester and SUT negotiate the direction?
- What is the corresponding Scheme program?



(ctd)

```
(let ((timeout (random)))
  (sync (input x (require (equal? x 'a))
                 ...)
        (wait timeout
           (sync (output 'B ...)))))
```

- What is the difference? Is there any?

Relevance of ioco theory

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

Conclusions

- Ioco is untimed, low-level theory based on LTSes
- Practical algorithms assume finite LTSes, which leads to the problem of abstraction

Testing of Concurrent Systems 2004

Lecture 19

30th Nov 2004

Course this far

14.9	1	Introduction, general concepts, traces
	2	Concurrent Scheme
21.9	3	Traces, specifications
	4	Seriality, execution introduction
28.9	5	Test steps and execution
	6	Test verdicts
12.10	7 8	Scheme programs as implementations, testing strategies, testers, and specifications
19.10	9	Conformance = trace inclusion
	10	Basic on-the-fly testing algorithm
26.10	11 12	Rational decision, "economics of testing"

2.11	13	Testing algorithms, planning, property covering
	14	State-space based algorithms
16.11	15	Symbolic state space exploration
	16	Symbolic execution of Scheme
23.11	17	Ioco theory
	18	Critique of ioco

Today: "advanced" topics

- Test script generation
- Combinatorial test case design
- TTCN-3
- FCT and software process
- Implementing a toy FCT tool

Test scripts

- Test script = explicitly given tester
- Usually assume reasonably efficient and executable implementation
- Our on-the-fly testing algorithm can be very slow
  - Planning, trace inclusion check, property coverage analysis take time

Solution

- Try to generate an explicit-form, fast test script and use it instead of the generic algorithm
  - Do trace inclusion checks for correctness afterwards offline
  - Compute plans and property coverage offline before execution
  - In general, this is partial evaluation

Partial evaluation

- Suppose we have a function definition

(define (f x y) …)

Partial evaluation (ctd)

- Let V be a certain value. Partial evaluation of f with respect to y := V produces a residual procedure g on on parameter x such that

(g x) = (f x V)

Partial evaluation (ctd)

- Thus, g is a specialization of f with respect to y := V. Similarly, f is a parameterized version of g, y being the "new" parameter.
- Partial evaluation is a well-known, advanced compiler technique.

The link

- The on-the-fly testing algorithm is parameterized by two series of data: algorithm's internal choices (e.g. choose a valid trace T), and SUT's choices that manifest as external behaviour (e.g. receiving a message from the SUT).

The link (ctd)

- In principle, the algorithm could have the signature

(define (otf-test internal-choices sut-behaviour) …)

- Now partial evaluate internal-choices out

Residual tester

- A residual tester would have the signature

(define (a-test-script sut-behaviour) …)

Practical notes

- There is a general way to do partial evaluation:

```
(define (g x)
  (let ((y V))
    <body of f>))
```

- But this is not interesting to us, because there is no categorical speed up
- The real problem is how to reap execution speed benefits from specialization

Combinatorial test case design

- Basic problem
  - A system has three parameters a..c
  - Every parameter has three potential values 1..3
  - We want to test the system's behaviour with different combinations of these parameters
  - There are $3^3 = 27$ full combinations

Pairing

- A common idea is not to test all the combinations, but to test a set of combinations such that every pair of any two values on two parameters has been tested

Example

Run	a	b	c
#1	1	1	1
#2	1	2	2
#3	1	3	3
#4	2	1	3
#5	2	2	1
#6	2	3	2
#7	3	1	2
#8	3	2	3
#9	3	3	1

General construction

- The idea can be generalized
- Define an arbitrary structure of partial parameter valuations to be covered
  - Explicit definition (enumerate the desired structures)
  - Implicit definition (use a language to define the structures)

Set cover

- In an explicit form, this is the <u>set cover</u> problem:
- Given a set X and a set Q of subsets of X, find a minimal/small subset S of Q such that

$$\cup S = X$$

Set cover example

- X = { “a=1, b=1”, “a=2, b=1”, …, “b=1, c=1”, “b=2, c=1”, … }
  - Every element of X is a pair to be covered
- Q = { { “a=1, b=1”, “b=1, c=1”, “a=1, c=1” }, { “a=2, b=1”, “b=1, c=1”, “a=2, c=1” }, …}
  - Every element of Q corresponds to a full valuation of the parameters a..c, and enumerates those pairs that the corresponding valuation covers

Set cover ctd

- Set cover is a NP-complete problem
- An approximation algorithm exists, but not very efficient
- In practice, more direct approaches can be used which avoid the explicit enumeration of the structures
- An instance of <u>combinatorial design</u>

Use within FCT

- E.g. parameter pair values can be used as properties to be covered
- An efficient property covering targeting on-the-fly testing algorithm would need to solve problems of this kind
- In practice can be also made a visible part of test design → classification tree method

TTCN-3

- "Testing and test control notation"
- Test programming language for telco systems standardized by ETSI
  - Also used in automotive industry and related segments today
- Original focus on protocols
  - Timers
  - Concurrency
  - Data template matching

TTCN-3 (ctd)

- Link to this class: conformance testing (in the telco way), testing of concurrent systems
- Formal conformance testing and TTCN-3 are not linked
- However, in theory test scripts generated from specifications can be rendered as TTCN-3 source code

Testing of Concurrent Systems 2004

Lecture 20

30th Nov 2004

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

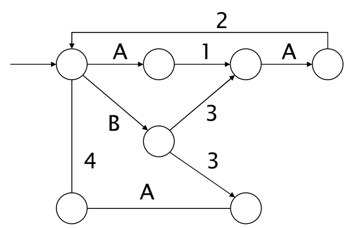
Process support

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

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

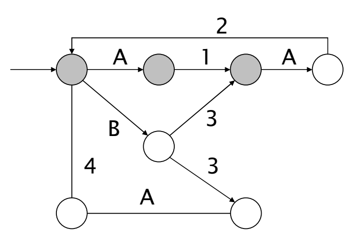


Step 1

- 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"

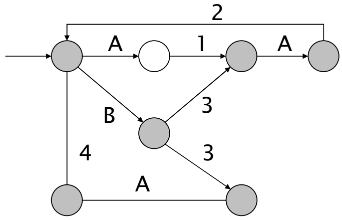
Example

"A1C3"



Example

“B3A4”

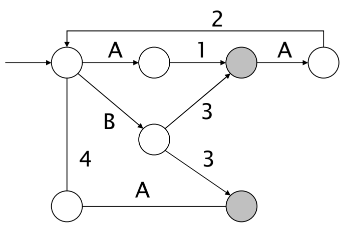


Step 2

- 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”



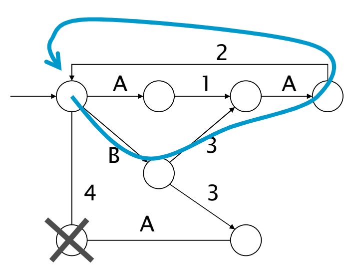
Step 3

- Build a test execution loop:
  - Check observed trace
  - Compute current specification states
  - Choose an input that is valid in all 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



Step 5

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

Next week

- Summary and conclusion
- Pre-examination (voluntary multiple-choice test, no effect on grade)