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
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**

Diagram:
- Tester
- SUT
- Interaction
- Specification
  - Guides
  - Defines correctness of

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**

Diagram:
- Tester
- SUT
- Message traffic
- Clock

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

- $\Sigma_{in}$ is the set of input messages
- $\Sigma_{out}$ is the set of output messages
- $\Sigma$ 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 $\Sigma$) + (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. <$\{\text{"hello" in, 0.5}\}$, 0.8>

Graphical sketch

Tester SUT

First input

Second input

First output

Trace end

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

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 an easy and clean dialect of LISP
- Scheme = interpreter for applicative order $\lambda$-calculus
- See e.g. Abelson & Sussman

Standard Scheme

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

```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

```scheme
(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:

```scheme
(sync
  (input <var> <body> …) …
  (output <expr> <body> …) …
  (read <point> <var> …) …
  (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

Queue
Producer thread
Consumer thread
Delay
A rendezvous point

Queued echo

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

(define (producer point queue)
  (if (empty-queue? queue)
      (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)

Specification

Guides

Tester

Interaction

SUT

Defines correctness of

Announces

Verdict = test result

Testing interface (replay)

Tester

Message traffic

SUT

Clock

Testing interface

A trace (replay)

<table>
<thead>
<tr>
<th>Event</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>First input</td>
<td>A</td>
</tr>
<tr>
<td></td>
<td>0.2</td>
</tr>
<tr>
<td>Second input</td>
<td>B</td>
</tr>
<tr>
<td></td>
<td>0.9</td>
</tr>
<tr>
<td>First output</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td>1.2</td>
</tr>
<tr>
<td>Trace end</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
</tbody>
</table>

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
- $\Sigma_{\text{in}}$ is the set of input messages
- $\Sigma_{\text{out}}$ is the set of output messages
- $\Sigma$ 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 $\Sigma$) + (nonnegative real number)
- Formally, set of events is $\Sigma \times [0, \infty)$
- E.g. <$\text{“hello world”}_{\text{in}}$, 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. <$\{<\text{“hello”}_{\text{in}}, 0.5>, 0.8>$

### Graphical sketch

<table>
<thead>
<tr>
<th>Tester</th>
<th>SUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>First input</td>
<td>A</td>
</tr>
<tr>
<td>t=0.2</td>
<td></td>
</tr>
<tr>
<td>Second input</td>
<td>B</td>
</tr>
<tr>
<td>t=0.9</td>
<td></td>
</tr>
<tr>
<td>First output</td>
<td>C</td>
</tr>
<tr>
<td>t=1.2</td>
<td></td>
</tr>
<tr>
<td>Trace end</td>
<td>......................... t=2</td>
</tr>
<tr>
<td>&lt;${&lt;A,0.2&gt;,&lt;B,0.9&gt;,&lt;C,1.2&gt;}$,2</td>
<td></td>
</tr>
</tbody>
</table>

### 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=$E,t>$ is a prefix of T’=$E’,t’$ (write T $\prec$ T’)
    - $t \leq t’$ and
    - $E = \{ <\alpha, \kappa> | <\alpha, \kappa> \in E’ \land \kappa < t \}$
Prefix sketch

Empty trace

Extensions

Prefix set

Snapshot

Snapshot example
Difference time

- Suppose $T$ and $T'$ are traces such that $T$ is not a prefix of $T'$ and $T'$ is not a prefix of $T$.
- $T$ and $T'$ are hence not equal.
- Define $\Delta(T, T') = \min t^* : T|_{t^*} \neq T'|_{t^*}$.

Specifications

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

Valid traces

- Every specification denotes a set of traces: the set of valid traces $\text{Tr}(S)$ is the set of valid traces for $S$.
- $\text{Tr}(S)$ must contain $\epsilon$.
- $\text{Tr}(S)$ must be prefix-complete.
- $\text{Tr}(S)$ must be serial.
Prefix-completeness

- A set $X$ of traces is prefix-complete if the following holds:
  - If $T \in X$ and $T' \preceq T$ 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 $<E', t+\delta> \in X$ such that $<E, t> \preceq <E', t+\delta>
  - Every trace of $X$ has at least one arbitrarily long extension in $X$

Motivation for seriality

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

- We assume there exists a countable set of implementations, denoted by \( I \).
- 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 \( \mu : S \rightarrow (I \rightarrow [0,1]) \).
- For every \( s \in S \), it holds that \( \sum_i \mu(s)[i] = 1 \).
- 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 \( T \) 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 \( T_1, T_2, T_3, \ldots \).
- Here \( T_1 \preceq T_2 \preceq T_3 \preceq \ldots \).
- These correspond to test steps.
- Many different trace sequences are possible.
- How do we formalize this?

Semantic function \( \xi \)

- 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.
### Signature
- Let $\mathcal{T}$ denote the set of all traces
- The signature is $\xi: I \times T \times \mathcal{T} \rightarrow (\mathcal{T} \rightarrow [0, 1])$

### 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

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

### 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 $T_1 \nleq T_2$, and $T_2 \nleq T_1$
- A technical convenience

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

- Trace probabilities
- Execution sketch
- Testing strategy
- Trace
- Probability distribution on
- Implementation
- Testing strategy
- Trace
- Probability distribution on

Testing of Concurrent Systems 2004

Lecture 5
28th Sep 2004

Course this far

- 14.9: Introduction
  - General concepts
  - Traces
- 21.9: Concurrent Scheme
- 4: Traces, specifications
- Serial, execution introduction

Traces and specifications

- Trace = set of events + end time stamp
  - Event = message + time stamp
  - Prefix, extension, snapshot
- Specification \( \cong \) set of valid traces
  - Prefix-closed
  - Serial
For all i, s and T, it must hold that
\[ \sum_{T'} \xi(i, s, T)[T'] = 1 \]

For all i, s, T and T' it must hold that
\[ \xi(i, s, T)[T'] > 0 \]

\[ \Rightarrow T < T' \]

Hence: every test step consumes time

Test step disjointness

For any i, s, T, and T1 and T2 it must hold that if T1 \neq T2 and
\[ \xi(i, s, T)[T1] > 0 \]
\[ \xi(i, s, T)[T2] > 0, \text{ then} \]
\[ T_1 \prec T_2, \text{ and} \]
\[ T_2 \prec T_1 \]

A technical convenience

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 = \langle E_i, t_i \rangle it holds that t_i < K.

Infinitely many test steps before time t=2
Choice of $\xi$

- We have defined properties of $\xi$, not the function itself
- The particular choice for $\xi$ depends on
  - the set of implementations $I$,
  - the set of testing strategies $T$, 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 $T_1 = <E_1,t_1>$, $T_2 = <E_2,t_2>$, ..., 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

This trace does not end a test step boundary

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 $P^*[i,s,T]$: $\max T_1,\ldots,T_n: \prod_{i\in[1,n-1]} \xi(i,s,T_i)[T_{i+1}]$ where $T_1 = \epsilon$ and $T_n = T$
- $P^*[i,s,T]$ is the compound probability for trace $T$, if $T$ "happens" at test step boundary

Sketch

$P^*[i,s,T] = 0.12$
Step 2: traces at non-boundaries

- Denote by $P[i,s,T]$

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

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

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

Execution summary

- $\xi$ defines execution semantics
- Properties for $\xi$
  - 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

- Specification
  - Guides
  - Defines correctness of
- Tester
  - Interaction
  - SUT
  - Announces
  - Verdict = test result

Verdicts

- Pass
- Fail
- Error
- Inconclusive
- Confused
Verdicts

- **Pass**
- **Fail**
- **Confused**
- **Inconclusive**

Correct execution Incorrect execution

Verdicts explained

<table>
<thead>
<tr>
<th>Verdict</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pass</td>
<td>System under test has behaved correctly</td>
</tr>
<tr>
<td>Fail</td>
<td>System under test has behaved incorrectly</td>
</tr>
<tr>
<td>Error</td>
<td>Tester has behaved incorrectly</td>
</tr>
<tr>
<td>Inconclusive</td>
<td>Pass, but a certain test purpose has not been met</td>
</tr>
<tr>
<td>Confused</td>
<td>Fail-and-Error, result produces by an ambiguous specification (a special corner case)</td>
</tr>
</tbody>
</table>

Verdicts as traffic lights

Pass Fail Error Inconclusive Confused

Calculating verdict

- Verdict is calculated from a trace $T$ and a specification $S$
  
  $\text{verdict}(T,S) \in \{ \text{pass}, \text{fail}, \text{error}, \text{conf} \}$

- No inconc, because requires a test purpose

Pass verdict

$\text{verdict}(T,S) = \text{pass}$

if and only if

$T \in \text{Tr}(S)$

Other verdicts

- Hence, $T \notin \text{Tr}(S)$ implies
  
  $\text{verdict}(T,S) \in \{ \text{fail}, \text{error}, \text{conf} \}$

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

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

Solution sketch (1)

Solution sketch (2)

Solution sketch (3)

Solution sketch (4)

Technicallity

- 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)
Details

- Assume \( T \notin \text{Tr}(S) \)
- Let \( V = \text{Tr}(S) \cap \text{Pfx}(T) \)
  - 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)

- \( \text{Tr}(S) = \cup \{ \text{Pfx}(<\{A, t'>, t>) \mid t \in [2, \infty), t' \leq 1 \} \)
- \( T = <\emptyset, 10> \)
- Note that \( T \notin \text{Tr}(S) \)
- \( V = \{ <\emptyset, t> \mid t \leq 1 \} \)
- \( K = [0, 1] \)
- Especially \(<\emptyset, 1>\) is in \( V \), because \(<\{A, 1>, 1.1>\) is valid

Details continued

- Choose \( \delta \in K \) (note: \( 0 \in K \) always, so \( K \) is not empty)
- For every \( T' \) in \( X_\delta \), \( T' \) differs from \( T \) and \( \Delta(T, T') \) is defined
- For every \( T' \), denote by \( \alpha \mid T |_{\Delta(T, T')} \) if not \( \tau \)
  - Otherwise denote by \( \alpha \mid T |_{\Delta(T, T')} \)
  - Note: \( \alpha \) cannot be \( \tau \)
- Let \( D_\alpha \) be the union of all \( \alpha \)
- \( D_\alpha \) lists those events on which valid extensions of \( T[\delta] \) differ from \( T \)

Example (open set)

- \( \text{Tr}(S) = \cup \{ \text{Pfx}(<\{A, t'>, t>) \mid t \in [2, \infty), t' < 1 \} \)
- \( T = <\emptyset, 10> \)
- Note that \( T \notin \text{Tr}(S) \)
- \( V = \{ <\emptyset, t> \mid t < 1 \} \)
- \( K = [0, 1) \)
- Especially \(<\emptyset, 1>\) 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 \( X_\delta \) denote the set of all valid extensions of \( T[\delta] \) beyond the end time stamp of \( T \)

Details continued

- Assume there exists \( \delta \in K \) such that \( D_\delta \subseteq \Sigma_{in} \)
  - Tester failure \( \rightarrow \) error
- Assume there exists \( \delta \in K \) such that \( D_\delta \subseteq \Sigma_{out} \)
  - SUT failure \( \rightarrow \) fail
- Otherwise
  - undefined \( \rightarrow \) confused
Details continued

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

Disjointness

- \( D_\delta \subseteq \Sigma_{\text{in}} \) and \( D_\delta \subseteq \Sigma_{\text{out}} \) are disjoint conditions, because
  - \( \delta \leq \epsilon \) implies \( D_\epsilon \subseteq D_\delta \)
  - \( D_\delta \) is always non-empty
  - \( \Sigma_{\text{in}} \) and \( \Sigma_{\text{out}} \) are disjoint

Summary

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

Testing of Concurrent Systems 2004

Lecture 7
12th Oct 2004

Course this far

- 14.9: Introduction
- 21.9: Scheme in everything
- 21.9: General concepts
- 21.9: Traces
- 21.9: Traces, specifications
- 28.9: Seriality, execution introduction
- 28.9: Test steps and execution
- 28.9: 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 $\xi$

Behaviour through $\xi$

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

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

▸ This is the set of all traces that are produced by some testing strategy with a non-zero probability
▸ Here we assumed that $\xi$ was defined for Scheme programs

Defining $\xi$

▸ We assume a definition of $\xi$ 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 $\Rightarrow$ 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 $T$ 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:

\[
\begin{align*}
&\text{(define (run)} \\
&\text{\quad (sync (input x (run))))} \\
&\text{\quad (run)}
\end{align*}
\]

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 $\rightarrow$ 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 $\xi$ (as usual)

Programs as full testers

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

Example

- Tester for Echo Program:
  \[
  \begin{align*}
  &\text{(begin} \\
  &\quad \text{(sync (output “hello”) } \\
  &\quad \text{\quad (sync (input x } \\
  &\quad \quad \text{\quad \quad \quad \quad \quad (if (equal? x “hello”) } \\
  &\quad \quad \quad \text{\quad \quad \quad \quad \quad \quad \quad \quad \quad (pass)}))} \\
  &\quad \quad \quad \text{\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad (wait 1 #f)))} \\
  &\text{\quad (fail))}
  \end{align*}
  \]

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:
  \[ VTr(p) = \{ <E,t> \mid <E,t> \in ETr(p) \land ERR \notin E \} \]
▶ 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):
  ```lisp
  (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 \( T = \langle \text{"hello"}_\text{in}, 1 \rangle, 3 \rangle \)
- Note: \( T \not\in \text{Tr}(p) \)
- What is \( \text{verdict}(T,p) \)?

Verdict

- Verdict is **ERROR**
- The set of valid prefixes of \( T \) is \( \{ \langle \emptyset, t \rangle | 0 \leq t \leq 1 \} \).
- Namely, \( \text{"hello"} \) 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 \( \text{"hello"} \) is the problem \rightarrow tester error

Use of require

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

Require trick

```scheme
(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 \( T = \langle \langle 3, 0 \rangle \rangle, 10 \rangle \)
- What should be \( \text{verdict}(T,p) \)??
- The solution is...

Require trick conclusion

- ... **FAIL**.
- Can you understand why?
Testing of Concurrent Systems 2004

Lecture 9
19th Oct 2004

Course this far

<table>
<thead>
<tr>
<th>14.9</th>
<th>1 - Introduction, general concepts, traces</th>
</tr>
</thead>
<tbody>
<tr>
<td>21.9</td>
<td>2 - Concurrent Scheme</td>
</tr>
<tr>
<td>28.9</td>
<td>3 - Traces, specifications</td>
</tr>
<tr>
<td></td>
<td>4 - Seriality, execution introduction</td>
</tr>
<tr>
<td>12.10</td>
<td>5 - Test steps and execution</td>
</tr>
<tr>
<td></td>
<td>6 - Test verdicts</td>
</tr>
<tr>
<td></td>
<td>7 - Scheme programs as implementations, testing strategies, testers, and specifications</td>
</tr>
</tbody>
</table>

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 testing strategy
  - a specific test execution trace
  - a specific tester
  - a specific tester execution

- 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 correct with respect to a specification S if for any implementation i: P[i, s, T] > 0 ⇒ verdict(T, S) ≠ 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

<table>
<thead>
<tr>
<th>Verdict</th>
<th>PASS</th>
<th>FAIL</th>
<th>ERROR</th>
</tr>
</thead>
<tbody>
<tr>
<td>PASS</td>
<td>Possible</td>
<td>Possible</td>
<td>Not possible (def. correct strategy)</td>
</tr>
<tr>
<td>FAIL</td>
<td>Not possible (synthetic testers)</td>
<td>Not possible (synthetic testers)</td>
<td>Possible</td>
</tr>
<tr>
<td>ERROR</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
</tr>
</tbody>
</table>

Trace map of a general system

- I: correct, producible behaviour
- II: incorrect, producible behaviour
- III: producible behaviour against malfunctioning environment only
- IV: correct behaviour not implemented
- V: incorrect behaviour not implemented
- VI: behaviour involving malfunctioning environment that has not been implemented

Correct and Incorrect Testers
- ETr(p)
**Trace map of a correct system**

**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 $ETr(i) = \{ T \mid \exists s \in T : P[i,s,T] > 0 \}$
- 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, $ETr(p, p') \subseteq Tr(p')$

**Conclusion**
- We have thus reduced the conformance of a program $p$ to a specification $p'$ to the equation $ETr(p, p') \subseteq Tr(p')$
- 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 directly 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 \( p \) is 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 \( \text{Tr}(p) \)
- \( \text{ETr}(p, p) \) is a subset of \( \text{Tr}(p) \)

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

\[
\begin{align*}
E &:= \emptyset, \text{clock} := 0 \\
\text{while } \{ \text{true} \} \text{ do} \\
X_e &:= \{ <E, \text{clock}+\epsilon> | \epsilon > 0, <E, \text{clock}+\epsilon> \in \text{Tr}(S) \} \\
X_{\text{in}} &:= \{ <E \cup <m, \text{clock}>, \text{clock}+\epsilon> | m \in \Sigma_{\text{in}}, \epsilon > 0, <E \cup <m, \text{clock}>, \text{clock}+\epsilon> \in \text{Tr}(S) \} \\
X_{\text{out}} &:= \{ <E \cup <m, \text{clock}>, \text{clock}+\epsilon> | m \in \Sigma_{\text{out}}, \epsilon > 0, <E \cup <m, \text{clock}>, \text{clock}+\epsilon> \in \text{Tr}(S) \} \\
N &:= X_e \cup X_{\text{in}} \cup X_{\text{out}} \\
\text{if } \{ N = \emptyset \} \text{ then } \text{FAIL} \\
\text{if } \{ \text{stopping criterion} \} \text{ then } \text{PASS} \\
\text{choose } T = <E', t> \text{ from } N \\
\text{if } T_{\text{clock}} \in \Sigma_{\text{in}} \text{ then } \text{send } T_{\text{clock}} : E := E \cup <T_{\text{clock}}, \text{clock}> \\
\text{wait for input until } t \text{ // } t > \text{clock} \\
\text{if } \{ \text{input m received at time } t' \text{ (clock } \leq t' < t) \} \\
\text{then } E := E \cup <m, t'>, \text{clock} := t' \\
\text{else } \text{clock} := t
\end{align*}
\]

Correctness arguments

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

Errors?

- Suppose the algorithm produces trace \( T \) such that \( \text{verdict}(T, S) = \text{ERROR} \)
- Hence \( T \notin \text{Tr}(S) \). 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 \( \epsilon \) 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 (T(S))
- Hence \text{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
- 21.9 2 - Concurrent Scheme
- 28.9 3 - Traces, specifications
- 19.10 4 - Seriality, execution introduction
- 12.10 5 - Test steps and execution
- 8 6 - Test verdicts
- 9 7 - Scheme programs as implementations, testing strategies, testers, and specifications
- 10 - Conformance = trace inclusion
- 10 - Basic on-the-fly testing algorithm

Economics of testing

- We know what testing is
- We know how we can test (at least basically)
- But why 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 without changing 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 rational choice is to choose the alternative that is most useful = has highest utility

Utility values

- Denote the utility of an alternative $\alpha$ by $u(\alpha)$; it is a real number
- Rational agent chooses alternative $\alpha_i$ 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 €) = \( \frac{1}{2} \) and L(2,000,000 €) = \( \frac{1}{2} \). 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
  \[ \mu : S \rightarrow (I \rightarrow [0, 1]) \]

Transforming distributions

- Assume S is a specification and \( \psi = \mu(S) \)
- Unknown SUT i is chosen according to \( \mu(S) \)—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 \( \psi' \)
- \( \psi \) is transformed to \( \psi' \) by s, T
- How?
- We will employ the Bayes' Rule

Bayes’ Rule

- The Bayes' Rule is a basic rule of conditional probability:
  \[ P(B|A) = \frac{P(A|B)P(B)}{P(A)} \]
- 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 \( \psi' \)

- We have produced trace T against unknown implementation with strategy s, a priori distribution of implementations being \( \psi \)
- The a priori probability for trace T is
  \[ \sum_i \psi[i] P[i,s,T] \]
- The a priori probability for implementation i is \( \psi[i] \)
- What is the a posteriori probability distribution for an implementation i?
Compute \( \psi' \) (2)

- \( \psi'(\psi,s,T)[i] = \frac{P[i,s,T] \psi[i]}{\sum_{i^*} (\psi(i^*) P[i^*,s,T])} \)
- 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 \( C = \sum_{i \in A(S)} \psi[i] \)
- The a posteriori probability for a correct system after testing is \( C' = \sum_{i \in A(S)} \psi'(\psi,s,T)[i] \)
- If testing results in \textit{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 \textit{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

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

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

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

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

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

- Summing up the previous series upto $k$, we get

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

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

$$(\sum_{0<i<k} 1 - F_i).$$

When $E$ does not converge

- What if $E$ is not converging?
- If $\lim n \to \infty F_n = 0$, 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 $(1 - F_0)/(1 - F_{n+1}) < \alpha < 1$ ($\alpha$ 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 \( B - \frac{F}{E} \)
▶ If \( B - \frac{F}{E} < 0 \), 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 \( \psi \) as usual:

\[
\sum_i \psi[i] (B - \frac{F}{E_i})
\]

▶ After testing, this becomes

\[
\sum_i \psi'[i] (B - \frac{F}{E_i})
\]

▶ But we can’t still quantify the benefit of testing...

... ctd ...

▶ ... because we do not know how long the system is used
▶ Assume life cycle of N steps
▶ Total increase of utility is

\[
N \sum_i (\psi'[i] - \psi[i]) (B - \frac{F}{E_i})
\]

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

Notes

▶ Analysis of this kind depends heavily on the failure model \( \mu \)
▶ How it is obtained?
▶ Can the calculations be carried actually out?
▶ What is the utility?
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]
    Xτ := { <E, clock+ε> | ε > 0, <E, clock+ε> ∈ Tr(S) }
    Xin := { <E∪<m, clock>, clock+ε> | m ∈ Σin, ε > 0, <E∪<m, clock>, clock+ε> ∈ Tr(S) }
    Xout := { <E∪<m, clock>, clock+ε> | m ∈ Σout, ε > 0, <E∪<m, clock>, clock+ε> ∈ Tr(S) }
    N := Xτ ∪ Xin ∪ Xout
    if [N = ∅] then FAIL
    if [stopping criterion] then PASS
    choose T = <E', t> from N
    if T|clock ∈ Σin 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

Choose valid continuation

none found → fail

Execute chosen continuation

stopping criterion → pass
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

\[
E := \emptyset, \text{ clock } := 0 \\
\text{ while } \{ \text{ true } \} \\
\quad \text{ Choose a suitable } G \text{ from } \text{Tr}(S) \text{ s.t. } <E, \text{clock} > \prec G \\
\quad X_e := \{ <E, \text{clock} + \epsilon > \mid \epsilon > 0, <E, \text{clock} > \prec G \} \\
\quad X_m := \{ <E \cup m, \text{clock} >, \text{clock} + \epsilon > \mid m \in \Sigma_m, \epsilon > 0, \\
\quad <E \cup m, \text{clock} >, \text{clock} + \epsilon > \prec G \} \\
\quad N := X_e \cup X_m \\
\quad \text{ if } \{ N = \emptyset \} \text{ then } \text{FAIL} \\
\quad \text{ if } \{ \text{ stopping criterion } \} \text{ then } \text{PASS} \\
\quad \text{ choose } T = <E', t> \text{ from } N \\
\quad \text{ if } T|\text{clock} \in \Sigma\text{ in} \text{ then } \{ \text{ send } T|\text{clock}, E := E \cup <T|\text{clock}, \text{clock}> \} \\
\quad \text{ wait for input until } t \text{ // } t > \text{clock} \\
\quad \text{ if } \{ \text{ input } m \text{ received at time } t' \text{ (clock } \leq t' < t) \} \\
\quad \quad \text{ then } E := E \cup <m, t'>, \text{clock} := t' + \epsilon \text{ for a "very" small } \epsilon \\
\quad \quad \text{ else } \text{clock} := t
\]

Abstract version

Choose target trace G (extension of the present trace)

Choose valid continuation that is a prefix of G

- none found
- fail
- Execute chosen continuation
- stopping criterion
- pass

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

\[ E := \emptyset, \text{clock} := 0, P := \emptyset \]

while [true]

\[ P := P \cup \text{UniversalPropertyCheck}(<E, \text{clock}>, S) \]

if [no G found]

Choose a suitable G from Tr(S) s.t. <E, clock> \(\prec\) G

\[ X_\tau := \{ <E, \text{clock}+\varepsilon> \mid \varepsilon > 0, <E, \text{clock}+\varepsilon> \preceq G \} \]

\[ X_{in} := \{ <E \cup <m, \text{clock}>, \text{clock}+\varepsilon> \mid m \in \Sigma_{in}, \varepsilon > 0, <E \cup <m, \text{clock}>, \text{clock}+\varepsilon> \preceq G \} \]

\[ X_{out} := \{ <E \cup <m, \text{clock}>, \text{clock}+\varepsilon> \mid m \in \Sigma_{out}, \varepsilon > 0, <E \cup <m, \text{clock}>, \text{clock}+\varepsilon> \preceq G \} \]

\[ N := X_\tau \cup X_{in} \cup X_{out} \]

if [N = \emptyset] then FAIL

if [stopping criterion] then PASS

choose \(T = <E', t>\) from N

if \(T|\text{clock} \in \Sigma_{in}\) then { send \(T|\text{clock} , E := E \cup <T|\text{clock}, \text{clock}> \)

wait for input until \(t\) // \(t > \text{clock}\)

if [input \(m\) received at time \(t'\) (\(\text{clock} \leq t' < t\))]

\[ E := E \cup <m, t'>, \text{clock} := t' + \varepsilon \text{ for a "very" small } \varepsilon \]

else \(\text{clock} := t\)

---

Abstract version

Choose target trace \(G\) aiming at new properties (extension of the present trace)

Choose valid continuation that is a prefix of \(G\)

Run \(G\) and check if it is found

Execute chosen continuation

Finish if \(G\) is found

A dive deeper

- How do we check if \(T \in \text{Tr}(S)\)?
- How do we compute the "properties" that a trace "necessarily" covers?
- How do we compute goal traces?

State space based computation

- The execution function \(\xi\) gives external behaviour, 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

\(\rightarrow\) 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 \(s' \in \text{next}(s)\), we write also \(s \rightarrow s'\)
### State spaces

- Assume we can associate with a specification
  - an initial state $s_0 = <c_0, \emptyset, 0>$
  - next state relation
- $\text{Tr}(S) = \{ T | \exists <c,T> : s_0 \rightarrow^* <c,T> \}$
- 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

```plaintext
W := \{s0\}
V := \emptyset
While W \neq \emptyset
  Choose <c,T> from W
  If T = T*
    Return FOUND
  Else if T \prec T*
    W := W \setminus \{<c,T>\}
    V := V \cup \{<c,T>\}
    W := W \cup (\text{next}(<c,T>) \setminus 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 $s \rightarrow_p s'$ if there is a transition from $s$ to $s'$ with properties $P$

### UniversalPropertyCheck($T^*, S$)

```plaintext
W := \{<s0, \emptyset>\}
V := \emptyset
P := \text{everything}
While W \neq \emptyset
  Choose <c,T>, P from W
  If T = T*
    P := P \cap P
  Else if T \prec T*
    W := W \setminus \{<c,T>, P\}
    V := V \cup \{<c,T>, P\}
    N := \{<s', P'> | s \rightarrow Q s', P' = P \cup Q\}
    W := W \cup (N \setminus 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>}
        EndIf
    EndIf
    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

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…

Discussion (ctd)

▶ … as mentioned on the “economics” lecture, it is impossible to prove that this would be a good thing
▶ So just a heuristic

Properties = coverage measures

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

Improvements

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

\[ W := \{ \alpha[s_0]\} \]
\[ V := \emptyset \]
While \( W \neq \emptyset \)

Choose \( s \) from \( W \)

If NotEmpty(\( s \ \ominus \ LiftTrace(T^*) \))

Return FOUND

Else

\[ W := W - \{ s \} \]
\[ V := V \cup \{ s \} \]
\[ N := \text{SymbolicSuccessors}(s) \ \ominus \ \text{LiftPrefix}(T^*) \]
\[ W := W \cup (N - V) \]

Return NOT FOUND

Comments

- \( \alpha \) maps a concrete state to a symbolic state representing the singleton set consisting of the concrete state
- \( \ominus \) 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 implemented?
- 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
Symbolic states

States

Symbolic states

Representation relation

Representation

- Let \( z \) be a symbolic state
- \( \gamma(z) \) is a set of states: the set of states represented 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
  \[ \text{Empty}(z) : \gamma(z) = \emptyset \]
- Intersection
  \[ \gamma(z \cap z') = \gamma(z) \cap \gamma(z') \]
- Subsumption relation
  \[ z \subseteq z' \Rightarrow \gamma(z) \subseteq \gamma(z') \]

Symbolic successors

- Next\((z) = \{ z' \mid z \rightarrow z' \} \)
- Axiom 2:
  \[ s \in \gamma(z), s \rightarrow s' \text{ implies } \exists z' \in \text{Next}(z) : s' \in \gamma(z') \]

Operations needed for symbolic trace inclusion check

- LiftTrace\((T) \)
  - Returns \( z \) such that
    \[ \gamma(z) = \{ s \mid \exists c : s = <c, T> \} \]
- LiftPrefix\((T) \)
  - Returns \( z \) such that
    \[ \gamma(z) = \{ s \mid \exists c, T' : s = <c, T'>, T' \preceq T \} \]
Symbolic trace inclusion check algorithm

\[
W := \{s_0\} \quad \text{and} \quad W := \emptyset
\]

While \( W \neq \emptyset \)

Choose \( z \) from \( W \)

If not Empty(\( z \cap \text{LiftTrace}(T^*) \))

Return FOUND

Else

\( W := W - \{z\} \)

\( V := V \cup \{z\} \)

\( N := \{ z'' | z' \in \text{Next}(z), z'' = z' \cap \text{LiftPrefix}(T^*) \} \)

\( W := W \cup (N - V) \)

Return NOT FOUND

Correctness discussion

- Suppose \( \gamma(z) \) are all reachable in the concrete state space

- Suppose \( z \rightarrow z' \)

- Then also \( \gamma(z') \) are all reachable by definition

- On the other hand, suppose \( s \) is reachable, and \( z \) is reachable such that \( \gamma(z) \) contains \( s \)

- Suppose \( s \rightarrow s' \)

- Then \( z' \) exists in the set \( \text{Next}(z) \) such that \( s' \in \gamma(z) \)

Symbolic states: example

\[\begin{align*}
\text{sync } (\text{input } x) \\
\text{sync } (\text{wait } 0.1) \\
\text{sync } (\text{output } (+ x 1) (\text{halt})) \quad \text{??}
\end{align*}\]

Discussion

- The symbolic state space depicts the whole infinite state space, but is finite in the example

- Individual states are represented symbolically as individual solutions to a constraint set

Constraint solutions

- Constraint set: \( \{X1>0, X3=X1+0.1, \text{number } X2, X4=X2+1\} \)

- \( X1=0.2, X3=0.3, X2=9, X4=10 \) 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!

- \( X2=\text{"hello"} \) does not lead to a solution

- Received value not number!

Computational point of view

- Constraint sets are easy to create, difficult 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

- $c = [t \rightarrow X1, x \rightarrow X2, ...]$
- $T = \{<X2_{in}, X1>, <X4_{out}, X3>, X3\}$
- $C = \{X1 > 0, X3 = X1 + 0.1, \text{number } X2, X4 = X2 + 1\}$
- $<<c, T>, C>$ 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 $<<c, T>, C \land Q(C')>$
- 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
- $<<c, T>, C> \cap <<?, T'>, C'>$:
  - match $T$ against $T'$, then yield $<<c, T>, C \land Q(C')>$
  - (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 \( C \Rightarrow C' \), check for the solvability of \( C \land \neg C' \)
- Assumes now that constraint sets are closed also under negation \( \Rightarrow \) 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

Scheme execution

Value stack

Term stack \( \langle \text{let} \ ((x \ (+\ 1\ 2)))\ (\text{sync}\ (\text{output}\ x))\rangle \)

Environment

Symbolic execution of Scheme

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

Scheme execution

Value stack

Term stack \( \langle (+\ 1\ 2)\ (\text{BIND}\ x)\ (\text{sync}\ (\text{output}\ x))\ (\text{RESTORE})\rangle \)

Environment
Scheme execution

Value stack

Term stack

Environment

Scheme execution

Value stack

Term stack

Environment

Scheme execution

Value stack

Term stack

Environment

Scheme execution

Value stack

Term stack

Environment

Scheme execution

Value stack

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 constraint variables as datums
Scheme execution

Value stack

Term stack: $x \cdot 5 \ (\text{APP} \ 2) \ (\text{TEST} \ (\text{gorblex} \ x) \ #\text{void}) \ (\text{RESTORE})$

Environment: $x := X1$

Scheme execution

Value stack

Term stack: $x \cdot 5 \ (\text{APP} \ 2) \ (\text{TEST} \ (\text{gorblex} \ x) \ #\text{void}) \ (\text{RESTORE})$

Environment: $x := X1$

Scheme execution

Value stack

Term stack: $5 \ (\text{APP} \ 2) \ (\text{TEST} \ (\text{gorblex} \ x) \ #\text{void}) \ (\text{RESTORE})$

Environment: $x := X1$

Scheme execution

Value stack

Term stack: $(\text{APP} \ 2) \ (\text{TEST} \ (\text{gorblex} \ x) \ #\text{void}) \ (\text{RESTORE})$

Environment: $x := X1$

Scheme execution

Value stack

Term stack: $(\text{TEST} \ (\text{gorblex} \ x) \ #\text{void}) \ (\text{RESTORE})$

Environment: $x := X1$

Constraints: $X2 \leftrightarrow (X1 > 5)$, number $X1$

Scheme execution

Value stack

Term stack: $(\text{gorblex} \ x) \ (\text{RESTORE})$

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 all potential internal choices of a correctly working SUT
- This yields to constraint solving over alternating quantifies (\(\exists\) 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

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

<table>
<thead>
<tr>
<th>States</th>
<th>Labels</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 2, 3, 4</td>
<td>a, b</td>
</tr>
<tr>
<td>Transition</td>
<td>(&lt;1, a, 2&gt;, &lt;2, a, 3&gt;, &lt;3, b, 4&gt;, &lt;4, \tau, 2&gt;, &lt;4, b, 1&gt;)</td>
</tr>
</tbody>
</table>

\(S_0 = 1\)

Traces

- The traces of an LTS are obtained by “walking” in it starting from the initial state, and collecting all symbols except \(\tau\)’s which denote “silent activity” and which are removed.

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 (≠ τ) 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 you cannot refuse an input and that you can’t deadlock.

Example

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

Not an IOTS

An IOTS

Testing Theory for IOTSs

- In the “ioco” testing theory, the Tester and the SUT are assumed to be IOTSs.
- Obviously, the Tester and SUT are mirror images of each other in the sense that outputs from SUT are inputs to Tester and vice versa.
- Hence, if L_O is the set of outputs from SUT, then this is the set of inputs to Tester, which must be always enabled in Tester.
- The 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
  \[ \text{obs}(t \mid i) \subseteq \text{obs}(t \mid s). \]
  (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 \( \text{obs}(L) \subseteq \text{obs}(L') \) if the subset relation holds for both the sets mentioned above.
- This leads to the input–output testing relation \( \leq_{\text{iot}} \). We write \( i \leq_{\text{iot}} s \) 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 \( \leq_{\text{iot}} \)...  
  - 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.
  - \( i \leq_{\text{iot}} s \) iff
    \[ \text{traces}(i) \subseteq \text{traces}(s) \text{ and } \text{Qtraces}(i) \subseteq \text{Qtraces}(s) \]
    where \( \text{Qtraces}(L) \) 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 \( \delta \) which denotes "I'm waiting for input" or "I'm quiescent".
Repetitive Quiescence (ctd)

- The name for $\delta$ is “suspension”.
- We call the traces of an IOTS with this extension (can produce $\delta$ when no output is possible) “suspension traces”, denoted by $\text{Straces}(L)$.

loco relation

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

What is the Difference?

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

Testing of Concurrent Systems 2004

Lecture 18
23rd Nov 2004

General comments

- loco 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 loco
- But realistic systems usually have infinite or very big state graphs
- Leads to the need to do manual abstraction
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?

Relevance of ioco theory

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

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
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 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 on-the-fly testing algorithm is parameterized by two series of data: the 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).

In principle, the algorithm could have the signature

\[
\text{define (otf-test internal-choices sut-behaviour) \ldots}
\]

Now partial evaluate internal-choices out

A residual tester would have the signature

\[
\text{define (a-test-script sut-behaviour) \ldots}
\]

There is a general way to do partial evaluation:

\[
\text{(define (g x) (let ((y V)) \langle body of f\rangle))}
\]

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

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

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

<table>
<thead>
<tr>
<th>Run</th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>#1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>#2</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>#3</td>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>#4</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>#5</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>#6</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>#7</td>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>#8</td>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>#9</td>
<td>3</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

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 set cover problem:
- Given a set $X$ and a set $Q$ of subsets of $X$, find a minimal/small subset $S$ of $Q$ such that
  \[ \bigcup S = X \]

Set cover example

- $X = \{ \text{"a=1, b=1", "a=2, b=1", \ldots, "b=1, c=1", "b=2, c=1", \ldots} \}$
  - Every element of $X$ is a pair to be covered
- $Q = \{ \{ \text{"a=1, b=1", "b=1, c=1", "a=1, c=1"}, \{ \text{"a=2, b=1", "b=1, c=1", "a=2, c=1"} \}, \ldots \}$
  - 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 combinatorial design

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 $\Rightarrow$ 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 (Æ 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

A1 2
A 1 3
B 3
1 2
A

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”
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.

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
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)