Welcome!

- This is T–79.5304: Formal Conformance Testing
- Lectures from 10 to 12 am, no regular tutorials
- 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: Managing Director 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 printed and distributed by the lecture note print service
Organization

- Lectured in Finnish
- All written material in English
- Examination material = lecture notes
- To pass the course you must pass the examination

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
  - (Often in the literature also “inconclusive verdict”—we return to this later)

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 (this simplifies the theory slightly)

Directions

- Input
  - input to the SUT
  - output from the tester
- Output
  - output from the SUT
  - input to the tester
- “SUT’s viewpoint”
Alphabets

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

Practical Matters

- 28th September there is no lecture because I am in Berlin
- Probably also no lecture on 26th October
- Consider ordering the lectures notes if not ordered yet
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

- UML statecharts and Java
- We have formal conformance testing tools for this choice, which is good

Zero–time execution principle

- We assume that all UML/Java execution consumes zero time
- The only exception are timeouts and explicit waiting for asynchronous communications
UML

▶ UML = Unified Modeling Language
▶ Standardized by OMG (Object Management Group)
▶ Current major version is UML 2

UML State Charts

▶ Most common state chart components:
  • States
  • Transitions
  • Initial and final states
Statechart Example

Initial state
This is where an object instance’s life begins.
State
State represents a control state. The object instance can “stay” in a state waiting for an event. (This is not true of initial and final states).

Final state
When an instance reaches its final state it finishes its behavior and dies.
Statechart Example

Transitions
Transitions can be “fired” then the instance is in the source state. Then the instance “moves” to the destination state.

Spontaneous transition
Spontaneous transition is a transition without any trigger. It is fired "spontaneously" when the instance has arrived in the source state.
Statechart Example

Trigger
A transition with a trigger is fired when the trigger “happens” and the instance is in the source state. Typically trigger is message reception.

Condition
A transition that has a condition is fired only if the condition evaluates to true.
Statechart Example

Action
A transition can have an action. It is code that is run when the transition is fired. In our case we write actions (and conditions) in Java.

Example

```java
system
{
    Inbound userIn : UserInput;
    Inbound netIn : SIPResp, SIPReq;
    Outbound netOut : SIPResp, SIPReq;
}

class SIPClient extends StateMachine {
    public int timeout = 1;
    public String dst = "sip:192.168.0.1";
}

record UserInput {
    public String input1;
}

record SIPResp {
    public int status;
}

record SIPReq {
    public String op;
    public String param;
}

void main() {
    a = new SIPClient();
    t = new Thread(a);
    t.start();
}
```
Formal Conformance Testing 2006

Lecture 5
5th Oct 2006
Course this far

1. Introduction
   - General concepts
   - Traces
2. Java + UML

FCT setup (replay)

Specification

Guides

Defines correctness of

Tester

Interaction

SUT

Announces

Verdict = test result
Testing interface (replay)

![Diagram showing tester and SUT connected by message traffic and clock]

A trace (replay)

```
First input
Tester       SUT
A            t=0.2

Second input
B            t=0.9

First output
C            t=1.2

Trace end
             t=2

<{<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

- $\Sigma_{\text{in}}$ is the set of input messages
- $\Sigma_{\text{out}}$ is the set of out 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. $<\text{“hello world”}_{\text{in}}, \ 1.4 \ \text{s}>$
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></th>
<th>Tester</th>
<th>SUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>First input</td>
<td>A</td>
<td>t=0.2</td>
</tr>
<tr>
<td>Second input</td>
<td>B</td>
<td>t=0.9</td>
</tr>
<tr>
<td>First output</td>
<td>C</td>
<td>t=1.2</td>
</tr>
<tr>
<td>Trace end</td>
<td></td>
<td>t=2</td>
</tr>
</tbody>
</table>

\(<\{<A,0.2>,<B,0.9>,<C,1.2>\},2>\)
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=<E,t> is a prefix of T’=<E’,t’> (write T ≼ T’) if
  - t ≤ t’ and
  - E = { <α, κ> | <α, κ> ∈ E’ ∧ κ < t }

Prefix sketch

<table>
<thead>
<tr>
<th>Tester</th>
<th>SUT</th>
<th></th>
<th>Tester</th>
<th>SUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>t=0.2</td>
<td></td>
<td>A</td>
<td>t=0.2</td>
</tr>
<tr>
<td>B</td>
<td>t=0.9</td>
<td></td>
<td>B</td>
<td>t=0.9</td>
</tr>
<tr>
<td>C</td>
<td>t=1.2</td>
<td></td>
<td></td>
<td>t=1.05</td>
</tr>
<tr>
<td></td>
<td>t=2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Empty trace

- $<\emptyset,0>$ is the empty trace, denoted by $\varepsilon$
- 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

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

Snapshot

- Denote $\Sigma_\tau = \Sigma \cup \{\tau\}$
- Here $\tau$ is an object that does not belong to set $\Sigma$
- Let $T = <E,t>$
- Assume $\kappa < t$
- Denote by $T|_\kappa$ the event at time $\kappa$, or $\tau$ if no event at trace $T$ has time stamp $\kappa$
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 of $T'$ and $T'$ is not a prefix of $T$
- $T$ and $T'$ are hence not equal
- Define
  \[ \Delta(T,T') = \min t^* : T|_{t^*} \neq T'|_{t^*} \]
Difference time sketch

Specifications

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

- $S$ 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 \textit{valid traces}
- If $S$ is a specification, $\text{Tr}(S)$ is the set of valid traces for $S$
- $\text{Tr}(S)$ must contain $\epsilon$
- $\text{Tr}(S)$ must be \textit{prefix-complete}
- $\text{Tr}(S)$ must be \textit{serial}
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 $\text{Tr}(S)$ must be prefix-complete?
Motivation for prefix-completeness

- \( \text{Tr}(S) \) 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?
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 $X$

Seriality sketch
Motivation for seriality

- Suppose non-serial Tr(S)
- There exists a valid trace T without an extension
- Let T’ ≼ T such that every far enough extension of T’ contains 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 $S$ denote the 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 $T_1 \preceq T_2 \preceq T_3 \preceq ...$
- 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

$\xi$ function properties

- Gives a probability distribution
- Test steps are proper trace extensions
- Progressivity (non-Zenoness)
- Test steps are disjoint
Signature

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

Execution

\[ \xi : I \times S \times \mathcal{T} \to (\mathcal{T} \to [0, 1]) \]
Gives probability distribution

- For all i, s and T, it must hold that 
  \[ \sum_{T'} \xi(i,s,T)[T'] = 1 \]
Test steps = proper trace extensions

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

For any i, s, T, and T₁ and T₂ it must hold that if T₁ ≠ T₂ and
ξ(i, s, T)[T₁] > 0 and
ξ(i, s, T)[T₂] > 0, then
T₁ ⊀ T₂, and
T₂ ⊀ T₁
A technical convenience

Execution sketch

A B

C

D

C

p=0.6

p=0.4

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

![Diagram showing infinitely many test steps before time \( t=2 \)]

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