Automatic Testing with Formal Methods

Christian Colombo

November 30th, 2010

Testing is Inevitable

- Can be applied to the actual implementation
  - Scales up
  - Can be applied to the actual implementation
  - No need to build a model of the system
    - It is complex to build a model
    - The system is a combination of software and hardware

The Testing Problem

- Test suite generation
- Test execution and behaviour observation
- Test oracle

The Challenges of Testing

- It involves a lot of effort to:
  - simulate the deployment environment
  - come up with a *good* test suite
  - run the tests
  - verify the tests
- Thus the need to automate these activities
  - Relatively easy to automate test execution and verification
  - Challenging to automate test case development

Testing and System Specification

- Testing verifies the system against a specification
- An incomplete/inaccurate/ambiguous specification hinders testing
- Test-driven development addresses this issue by forcing programmers to write their tests before coding
  (Forcing them to write a low-level specification)
- Formal specifications are unambiguous and can be processed automatically

Different Types of Testing

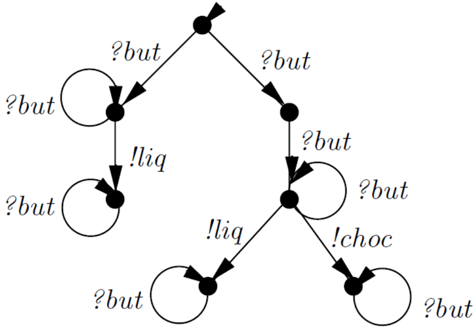
- Aspect to be tested
  - Functionality
  - Reliability
  - Availability
  - Robustness
  - Load
- Level of abstraction
  - Unit
  - Integration
  - System
- Levels of system visibility
  - White box
  - Grey box
  - Black box

Model-Based Testing

- Black box, functional testing
  . . . *conformance (w.r.t specs) testing*
- Use a model of the system to intelligently test it:
  - Guide test-case generation
  - As an oracle of the test results

Testing Reactive Systems

- Reactive systems continually react to stimuli from the environment
- Examples: embedded systems and protocols
- Generating tests on-the-fly (while executing) is beneficial as the test can be arbitrarily long



- $s \xrightarrow{a} s'$ — when the system is in state s, it may perform interaction a and progresses to state s'
- $r_1 \overset{?but \cdot ?but \cdot !choc}{\Longrightarrow}$ — the labelled transition system can produce chocolate after two button presses
- [TB99]

Input/Output

- A set of input actions L_I: $\{?a, ?b, \ldots\}$
- A set of output actions L_U: $\{!a, !b, \ldots\}$
- With all inputs enabled at each state:
  $\forall s \in S, ?a \in L_I \cdot \exists s' \in S \cdot s \xrightarrow{?a} s'$

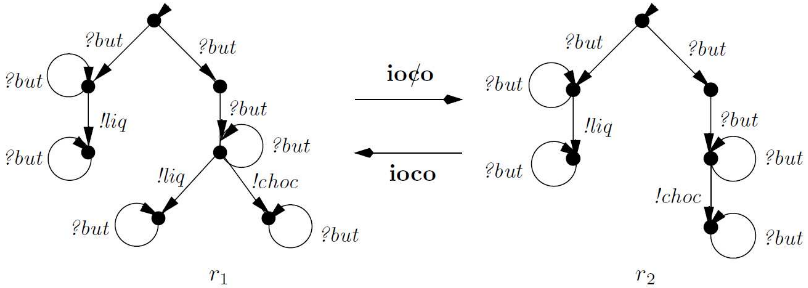
Definition of Conformance

- Let i represent an input/output transition system and s a specification in terms of a labelled transition system
- $s \; after \; \sigma \stackrel{\text{def}}{=} \{s' \mid s \stackrel{\sigma}{\Longrightarrow} s'\}$
- $out(s) \stackrel{\text{def}}{=} \{a \in L_U \mid s \stackrel{a}{\longrightarrow}\} \cup \{\delta \mid \forall a \in L_U : p \stackrel{a}{\not\longrightarrow}\}$
- $out(s \; after \; \sigma)$ — all outputs possible when consuming σ starting from s
- $L = L_I \cup L_U \cup \{\delta\}$
- $i \; ioco \; s \iff \forall \sigma \in L^* \cdot out(i \; after \; \sigma) \subseteq out(s \; after \; \sigma)$

Definition of Conformance

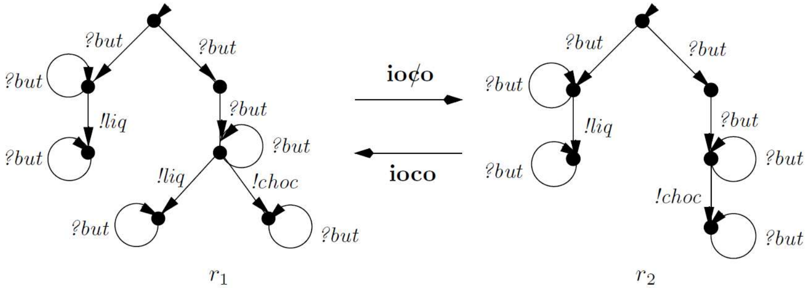
- Let i represent an input/output transition system and s a specification in terms of a labelled transition system
- $s \; after \; \sigma \stackrel{\text{def}}{=} \{s' \mid s \stackrel{\sigma}{\Longrightarrow} s'\}$
- $out(s) \stackrel{\text{def}}{=} \{a \in L_U \mid s \stackrel{a}{\longrightarrow} \} \cup \{\delta \mid \forall a \in L_U : p \stackrel{a}{\not\longrightarrow} \}$
- $out(s \; after \; \sigma)$ — all outputs possible when consuming σ starting from s
- $L = L_I \cup L_U \cup \{\delta\}$
- $i \; ioco \; s \iff \forall \sigma \in L^* \cdot out(i \; after \; \sigma) \subseteq out(s \; after \; \sigma)$
  i implements s if in any situation it never produces an output not produced by the specification s.

Example



- r_2 *ioco* r_1 but not r_1 *ioco* r_2

Example



- r_2 *ioco* r_1 but not r_1 *ioco* r_2
- $!choc \in out(r_1 \ after \ ?but \cdot \delta \cdot ?but)$ and $!choc \notin out(r_2 \ after \ ?but \cdot \delta \cdot ?but)$

The Perfect Test Suite

- Detects all ioco-erroneous implementations . . .

The Perfect Test Suite

- Detects all ioco-erroneous implementations (completeness)

The Perfect Test Suite

- Detects all ioco-erroneous implementations (completeness)
- Detects only ioco-erroneous implementations . . .

The Perfect Test Suite

- Detects all ioco-erroneous implementations (completeness)
- Detects only ioco-erroneous implementations (soundness)

The Perfect Test Suite

- Detects all ioco-erroneous implementations (completeness)
- Detects only ioco-erroneous implementations (soundness)
- Given a spec. s, an implementation i
- Test suite T_s generated by algorithm T on spec s

The Perfect Test Suite

- Detects all ioco-erroneous implementations (completeness)
- Detects only ioco-erroneous implementations (soundness)
- Given a spec. s, an implementation i
- Test suite T_s generated by algorithm T on spec s
- The perfect algorithm would have that:

$$\forall i, s: \; i \; ioco \; s \iff test\_exec(T_s, i) = pass$$

A Practical Test Suite

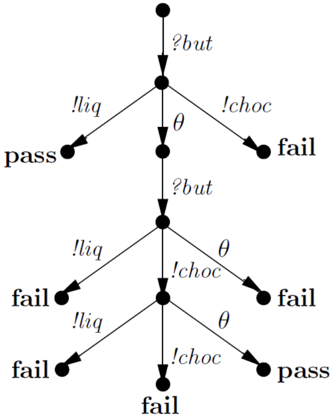
- In practice it is not feasible to have a sound and complete test suite
- Therefore we at least need soundness... if a test fails, then we are sure the implementation is incorrect

Test Cases

- A test case is a labelled transition system (lts) with a special structure:
  - A finite and tree-structured lts
  - each terminal state is either pass or fail
  - for each non-terminal state, there is either:
    - a transition labelled with a system input

      or
    - a transition for each system output and another with θ (a timeout)

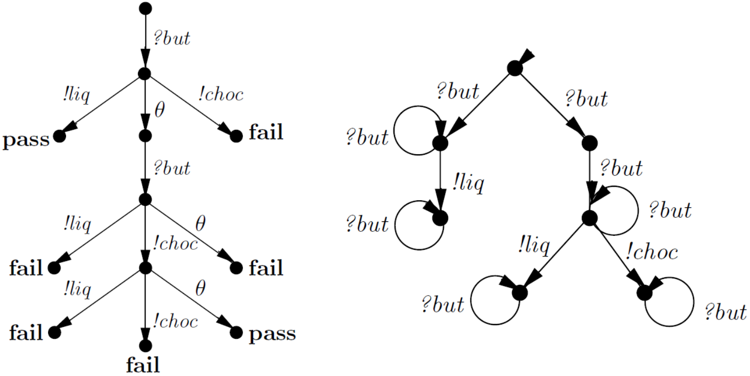
Test Case Example



Test Case Execution

- Executing a test case involves:
  - Executing the test case and the implementation simultaneously
  - If the test case lts ends in a failure, then the *fail* verdict is assigned...
  - ... and vice-versa if the test case succeeds

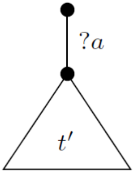
Test Case Execution Example



- Executing both lts' simultaneously may result in $?but \cdot \theta \cdot ?but \cdot !liq$
- Leading to fail

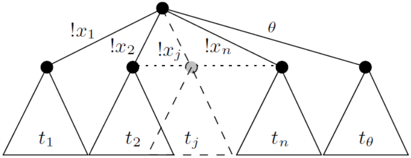
Test Derivation Algorithm

- s is a lts specification with initial state s_0
- S is a set of states in which the implementation can be in at a particular stage of the test case
- A test case t is obtained from s by applying one of the following non-deterministic choices
  1. $t :=$ • *pass*
  2. $t :=$



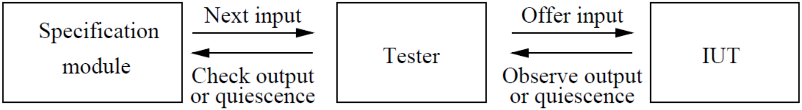
- Try all possible outputs and check which would signify a failure.

$t \quad :=$



where $L_U = \{x_1, x_2, \ldots, x_n\}$, $1 \leq j \leq n$:

if	$x_j \notin out(S)$	then	$t_j = \textbf{fail}$
if	$\delta \notin out(S)$	then	$t_\theta = \textbf{fail}$
if	$x_j \in out(S)$	then	t_j is obtained by recursively applying the algorithm for S **after** x_j
if	$\delta \in out(S)$	then	t_θ is obtained by recursively applying the algorithm for $\{s \in S \mid s \xrightarrow{\delta}\}$.



- Test inputs and outputs are generated lazily... step by step (as in the algorithm described above)
  - either the tester decides to generate a stimulus to the implementation under test (IUT)

    or
  - the tester observes the output produced by the IUT

Advantages of Testing with Formal Methods

- Reduce ambiguity in specifications
- Automatic maintenance of tests
- Arbitrarily long tests generated lazily

Disadvantages of Testing with Formal Methods

- Random testing instead of manually selected test cases
- Steep learning curve
- High initial costs to come up with formal specifications

References

G. J. Tretmans and A. F. E. Belinfante.
Automatic testing with formal methods.
In *EuroSTAR'99: 7th European Int. Conference on Software Testing, Analysis & Review, Barcelona, Spain*, Galway, Ireland, 1999. EuroStar Conferences.