Testing Against Requirements Using UML Environment Models

Denis Hatebur, Maritta Heisel
{denis.hatebur, maritta.heisel}@uni-due.de

Thomas Santen and Dirk Seifert
{santen,seifert}@cs.tu-berlin.de,
dirk.seifert@loria.fr

Universität Duisburg-Essen
Fakultät Ingenieurwissenschaften, Fachgebiet Software Engineering

Technische Universität Berlin
Fakultät Elektrotechnik und Informatik, Fachgebiet Softwaretechnik

LORIA – Université Nancy 2, France
ITESYS Institut für technische Systeme GmbH, Dortmund

June 5, 2008
Problem

- model-based software development: set up models of software systems to be constructed
- models should not be used for code generation as well as for testing

Our Approach

- test against requirements, not against specification
- explicitly model the environment in which the software will operate
- use UML state machines with tool support
- consider reactive/embedded systems
On-the-fly testing
- generating and executing test cases are intertwined
  + no state explosion
  - can start if implementation is available

Batch testing
- test cases are generated and stored for later execution
  + early test preparation possible
  - all possible behavior variants must be computed
<table>
<thead>
<tr>
<th>Terminology (Jackson) I</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Machine</strong></td>
</tr>
<tr>
<td><strong>Environment</strong></td>
</tr>
<tr>
<td><strong>System</strong></td>
</tr>
<tr>
<td><strong>Requirements</strong></td>
</tr>
<tr>
<td><strong>Specification</strong></td>
</tr>
</tbody>
</table>
Domain knowledge \textit{indicative} statements; consist of facts and assumptions:

- Facts describe what holds in the environment, no matter how we build the machine.
- Assumptions describe things that cannot always be guaranteed, but which are needed to fulfill the requirements, e.g., rules for user behavior.

Domain knowledge is needed to derive specifications from requirements: $S \land D \Rightarrow R$, $D \equiv F \land A$
Running Example: SunBlindControl

The sunblind can manually be lowered or pulled up.

The sunblind is automatically lowered on sunshine for more than one minute.

The sunblind should not be destroyed by heavy wind.

The environment consists of user, sunblind, sun and wind.
Transforming a Requirement into a Specification

$R_1$ The sunblind is not destroyed by wind.

$F_1$ Heavy wind for more than 30 sec is destructive.

$A_1$ Heavy wind for less than 30 sec is not destructive.

$F_2$ If the sunblind is up, it cannot be destroyed by wind.

$R_1'$ The sunblind is up if there is heavy wind for more than 30 sec.

$F_1 \land A_1 \land F_2 \land R_1' \Rightarrow R_1$
Transforming a Requirement into a Specification II

\(F_3\)  It takes less than 30 sec to pull up the sunblind.

\(R_1''\)  If there is heavy wind and the sunblind is not up, it is pulled up.

\(F_3 \land R_1'' \Rightarrow R_1'\)

\(F_4\)  There is heavy wind if and only if the wind sensor generates more than 75 pulses per sec.

\(F_5\)  Turning the motor left pulls up the sunblind.

\(S_1\)  If the wind sensor generates more than 75 pulses per sec and the last signals to the motor have not been *turn left*, followed by *motor left blocked* and *stop motor*, then the *turn left* signal is sent to the motor.

\(F_4 \land F_5 \land S_1 \Rightarrow R_1''\)

\(F_1 \land F_2 \land F_3 \land F_4 \land F_5 \land A_1 \land S_1 \Rightarrow R_1\)
Testing against the specification

- Does the machine generate the *turn left* signal?
- If the specification was not correctly derived from the requirements, the SUT passes the test nevertheless.

Testing against the requirements

- Does the sunblind enter a state where it would be destroyed?
- Detects errors made in transforming requirements into specifications.
- Checks if customer needs are satisfied (acceptance test).
Testing with Environment Models

- the environment is modeled using UML state machines
- this model explicitly contains the facts and assumptions about the environment
- the environment model consists of the input event generator and adapters:
  - the input event generator produces abstract events
  - adapters transform abstract events such as pull up sun blind into concrete ones, such as turn motor left.
- the requirements are expressed as state machines that check whether a requirement is violated
CO: Concrete Observation
AO: Abstract Observation
CS: Concrete Stimulus
AS: Abstract Stimulus
tick: Request for new Abstract Stimulus
Violation: Test Result
“On the Fly” - Test System Architecture: Sun Blind

State Machine Executor

Requirements

Environment Model

Input – Event – Generator

Adapters

System Under Test

CO: Concrete Observation
AO: Abstract Observation
CS: Concrete Stimulus
AS: Abstract Stimulus
tick: Request for new Abstract Stimulus
Violation: Test Result

lowerSunBlind, pullUpSunBlind, stopSunBlind, ...
turnMotorRight, turnMotorLeft, stopMotor, ...
windPulse, motorLeftBlocked, ...

Denis Hatebur
Testing Against Requirements Using UML Environment Models
Pattern for Requirements and Environment Model

Environment and Requirements State Machine

R1

R2

Rn

Input–Event–Generator

Adapter
Assumption A1 is modeled explicitly. Probabilities are given for the different transitions.
The adapter transforms the abstract stimuli into concrete ones.
Adapter: Concrete to Abstract Observations

The adapter transforms concrete observations into abstract ones.

Motor_Adapter

<<uses>>

WaitForMotorSig

<<provides>>

<<interface>>

abstr_motor_event

stopSunBlind ()
pullUpSunBlind ()
lowerSunBlind ()

<<interface>>

concr_motor_event

stopMotor ()
turnMotorRight ()
turnMotorLeft ()

<<interface>>

Motor_Adapter

stopMotor () / stopSunBlind ()

turnMotorLeft () / pullUpSunBlind ()

turnMotorRight () / lowerSunBlind ()
The postcondition of the requirement is checked when all preconditions are fulfilled.

“When [event$R_i$] happens, [controlled domain] should be in [desiredState$R_i$].”
Requirement: Example R1

If there is destructive wind, it must be checked if the sunblind is up. If yes, the requirement is satisfied. Otherwise, it is violated. The *Fail* state corresponds to a state where the sunblind would be destroyed.
The test case generator simulates the SUT in its environment and stores the resulting test cases.

The test execution unit executes the test cases on the SUT.
Testing Against Requirements Using UML Environment Models

Denis Hatebur
**TEAGER**

- research prototype
- clarifies UML state machine semantics
- executes imported UML state machines (for model validation and on-the-fly testing)
- generates test cases according to imported UML state machines (for batch testing)
  - probabilistic trigger selection
  - computes a complete behavioral model for a given search depth to determine the expected behavior of the SUT
- executes generated test cases
- LOCs: some 18k
Summary

Approach

- based on Jackson’s terminology
- uniform architecture for testing of reactive/embedded systems
- requirements are modeled explicitly with state machines using patterns
- facts and assumptions are modeled in the input-event generator or in the adapter state machines
- once these models have been set up manually (but systematically), the tests are performed automatically, using the tool TEAGER

Benefits

- environment models allow testing against requirements
- modeling patterns make approach systematic
- on-the-fly as well as batch testing is supported
- tool-support makes environment-based testing practical