Automated Model-Based Testing of Embedded Real-Time Systems

Jan Peleska
jp@tzi.de

University of Bremen

Bieleschweig Workshop 7
2006-05-05
Outline

▶ Objectives
▶ Basic concepts and definitions
▶ Conventional versus model-driven development and test cycle
▶ Overall Goals for Testautomation
▶ Resulting Major Work Packages
▶ Solutions: Intermediate model representation
▶ Solutions: Test case/test data generation
▶ Solutions: Tools chain
▶ Test case generation and test oracles for reactive real-time systems
▶ Test data generation
▶ Conclusion and ongoing research
Objectives

Methods, software tools and hardware test benches for

- Automated test case generation,
- Automated test oracles (= Checkers for SUT behaviour),
- Automated test data generation

for reactive real-time systems

SUT = System Under Test
Basic concepts and definitions (1)

- Trustworthy test system – sound and complete:
  - For each deviation of SUT behaviour from its specification, a test case uncovering this behaviour will finally be executed
  - System under test (SUT) which complies with its specification is never rejected in a test execution

- Combinational systems: Behaviour is specified by a Boolean mapping from inputs to outputs – no dependencies on internal state

- State-based real-time systems: Behaviour depends on internal state – one or more clocks (timers) are managed to control timed input/output behaviour

- Hybrid systems: Both time-discrete and time-continuous observables are managed
Basic concepts and definitions (2)

- **Specification-Based (Black-Box) Testing.** Test cases, associated data and checkers against expected results are derived from system-/software-/module- requirements specification.

- **Structural (White-Box) Testing.** Test cases and associated data are derived from
  - Code control structure (if-else, switch, while) of isolated methods (functions, procedures)
    - Main focus in the railway domain: decision coverage (C1)
    - Main focus for avionic software: multiple condition/decision coverage (MCDC)
  - Control coupling among methods (f() calls g())
  - Control coupling among tasks (P sends message to Q)
  - Data coupling (method accesses global data)
Basic concepts and definitions (3)

Examples for specification formalisms:

- **Composite structure diagrams** – architecture, including interfaces and parallelism
- **Class diagrams**
- **UML2.0 Statecharts** – extended by state invariants and flow conditions about time-continuous behaviour in the HybridUML profile
- **Temporal logics or trace assertions** – implicit assertions about admissible I/O sequences and state transitions
- **Method bodies specified in**
  - **OCL** – implicit assertions about method functionality
  - **C/C++, Java etc.** – explicit “low-level specification”
  - **Pseudocode** – explicit “higher-level specification”
- **Object diagrams** and **Configuration data** (＝“Projektierung”) describing concrete instances of generic systems
Basic concepts and definitions (4)

- Test levels
  - Unit Test (= Module Test)
  - Software Integration Test (SWI-Test)
  - Hardware-/Software Integration Test (HSI Test)
Conventional Development & Test Cycle

1. Software Requirements
2. Architectural SW Design
3. Module Design
4. Code
5. Functional Module Test Cases
6. Functional/Robustness Test Cases
7. SW & HW/SW-Integration Test (Functional)
8. SW Integration Test (Structural)
9. Module Test Functional & Structural
Model-Driven Development & Test Cycle

- PIM (Platform Independent Model)
- PSM (Platform Specific Model)
- Code
- Functional/Robustness Test Cases
- Functional Module Test Cases
- SW Integration Test (Structural)
- SW&HW/SW-Integration Test (Functional)

Automatic, Semi-automatic, Automatic
Conventional versus Model-Based Testing (1)

Merits of the conventional development & test cycle:

- Manual activities performed by independent experts introduced redundant "specifications": test cases were developed as partial functional requirements
- This redundancy helped to validate the specifications
- Manually developed structural tests helped to verify the architecture and code structure
Conventional versus Model-Based Testing (2)

For model-driven development & testing, the redundancy has to be introduced in the model, because test cases and associated data are derived in an automatic way:

- **Functional/architectural Model Level:** Temporal logic, trace logic in addition to behavioural and structural models (e.g. Statecharts, Composite Structure Diagrams)
- **Module Level:** Implicit specifications (e.g. OCL) in addition to explicit code
- Most of these consistency checks can be performed on model level, without executing the generated code!, **but** . . .
- . . . the implicit specifications will hardly ever be complete – so correctness of checks does not imply model correctness!
The consequence for the model-driven approach: Two focal points for testing in the future,

- **Testing on model level** – software tests in simulation environment – complemented by model checking and automated theorem proving: Detect logical failures

- **Hardware-in-the-Loop testing** on HW/SW integration and system integration level: detect
  - Failures in interface handling
  - Failures in arithmetics on target processors
  - Failures in memory management
  - Failures in real-time behaviour
Testing the code against the original model is of minor importance, since mature code generators are / will become highly reliable.

The model will always remain an approximation of the real target environment!
Currently, our main goals and recently completed developments are:

1. Automated Test Case Generation for Specification-Based Testing
   - Test Case $\equiv$ formal specification of situation to be tested – uses data abstraction
   - Objective (1.1): generate test cases from abstract specification models – such as HybridUML, Timed CSP
   - Objective (1.2): make generation mechanisms easily available for variety of specification formalisms
   - Objective (1.3): Provide mechanisms to influence automated test case generation by available expert knowledge
Overall Goals for Testautomation (2)

2. Automated Test Data Generation

- Test data \(\equiv\) concrete data to be used at interfaces to/from SUT and for checking SUT reactions against expected results
- Objective (2.1): generate concrete test data from abstract test cases – data refinement
- Objective (2.2): generate concrete test data for structural testing of code
- Objective (2.3): make generation mechanisms of (2.2) easily available for different programming languages – C, C++, Java, Assembler

Observation: (2.1) is of currently of minor importance since Case Tools mostly use concrete programming languages to describe events, conditions and actions
Overall Goals for Testautomation (3)

3. Automated Test Oracles

- Check SUT behaviour when executing test cases with concrete test data
- Checks have to be performed against one of the following types of specifications:
  - Explicit specification model which is assumed to be correct – for example validated beforehand by means of model checking
  - Redundant explicit specification model generated by the test team
  - Implicit specifications:
    - Temporal logic formulae
    - Trace logic
    - Pre-/post conditions
    - Regular expressions
Resulting Major Work Packages (1)

- **WP01: Intermediate representation model RT-Tester IRM**
  - IRM ≡ an **infrastructure** for test generation and test evaluation algorithms
  - Based on transition systems represented as collections of graphs
  - May consist of various sub-models

- **WP02: Transformers to IRM**
  - HybridUML, Timed CSP, Scade, C, C++, Java, Assembler → IRM

- **WP03: IRM symbolic simulator IRM-SYSI**
  - Symbolic expression evaluation
  - Determination of coverage achieved for given input data and timing behaviour
Resulting Major Work Packages (2)

- **WP04: IRM loop expansion generator IRM-LEG**
  - Expands cycles in the IRM until fixpoint is reached

- **WP05: IRM path condition generator IRM-PCG**
  - Create logical expressions for data and timing conditions suitable for covering a path through the IRM

- **WP06: IRM composer IRM-COP**
  - Combines different IRM sub models for global test generation
  - Delegates sub-tasks of the global test data generation to sub-models and their generators
Resulting Major Work Packages (3)

- **WP07: PCG solver collection IRM-SLV**
  - Determines which concrete input data and timing conditions to be used, in order to cover a specific path in the IRM
  - Observe that the general problem requires *constraints solving* for non-linear mixed Boolean-Discrete-Time-continuous problems:
    
    ```
    if ( a and not(b or c) and (i1 < 3*i2*i3) 
      and (expf(x@now  * y@now) > 
          x@(now-delta)*expf(y@(now-delta)) ) ) {
        ...
    }
    ```
  - Enforces test data generation heuristics like
    - Boundary value testing \(\equiv\) constraint solving with different objective functions to maximise
    - Data generation from different equivalence classes – each class defined by different constraint
Resulting Major Work Packages (4)

- **WP08: Path generators IRM-PG**
  - Construct specific paths through the IRM, in order to fulfil certain coverage or correctness criteria with the generated test data
  - **PG0:** path generator to achieve state coverage
    - Statement coverage C0 in code coverage testing
    - Unknown meaning in specification-based testing?
  - **PG1:** path generator to achieve transition coverage
    - Multiple condition/decision coverage MC/DC in code coverage testing
    - Requirements coverage in specification-based testing
  - **PG2:** (Timed) W method and derivatives
    - To demonstrate model equivalence in specification-based testing
    - Would exercise every possible transition from every possible program state in code coverage testing – no practical application possible here?
Resulting Major Work Packages (5)

- WP08: Path generators IRM-PG (continued)
  - Statistic path generation (M.-C. Gaudel) ≡
    - Guarantees good distribution of a random selection of paths to be exercised in the tests when exhaustive path coverage is impossible
Resulting Major Work Packages (6)

- WP09: Data Flow Analyser IRM-DFA
  - Helps to determine which objects influence the path selection performed by the SUT
  - Avoids unnecessary expansion of loops in the IRM

- WP10: SAT or/and OBDD solver
  - Helps to identify logically impossible solutions before starting constraint solver
  - May generate the test data for Boolean or mixed Boolean/integer problems
Resulting Major Work Packages (7)

▶ WP11: Stub Generator accepting behavioural specifications
  ▶ Generates stubs from
    ▶ specification of input parameter constraints ("expected input values") for each stub call in a sequence
    ▶ specification of explicit return values / writes to reference values for each stub call in a sequence
    ▶ specification of constraints referring to (1) position of stub call in the call sequence, (2) logical formulae referring to input and output parameters and global variables

▶ WP12: Assertion evaluator
  ▶ Allows to specify assertions with pre-/post conditions and temporal constraints
Solutions: Intermediate Model Representation

- A hierarchy graphs representing
  - Statechart levels
  - Control flow of methods
- Tree structures specifying hierarchic dependencies
- Intermediate code representation ("3 address code") for sequential algorithms – conditions, actions, $\Delta t$ integration of flows
Solutions: Intermediate Model Representation

- Constraints attached to graph nodes and edges representing
  - Transition conditions
  - State invariants
  - Flow conditions
- Tree structures representing “active” constraints
Part 1 – Model-based simulation and testing

INVARIANT: not CAB_PRESS_LOW

Lighting_scenario_on

INVARIANT: intensity < max
FLOW: d intensity/dt = 0

Lighting_scenario_off

INVARIANT: intensity < m2
FLOW: d intensity/dt = f(t)

Flow1:

Lighting_scenario_on/t0=now; activate(Flow1);

Lighting_scenario_off/suspend(Flow1);

Intermediate model

Solution to differential equation:

Flow1:

intensity(t) = intensity(t0) + \int f(s)ds
Part2 – Large scale simulation

Solutions:

- **Multi-threading architecture** with high-speed context switching in user space
- **Abstract Machines** encapsulate state-based sequential simulations as threads
- **Flows** encapsulate $\Delta t$-integration steps of time-continuous data changes as threads
Solutions: Test Case/Test Data Generation

A layered architecture of interacting generation algorithms

- Statechart traversal for generation of abstract timed traces
- Interval analysis for abstract interpretation of sequential algorithms and general constraint handling
- Differential geometric interpretation for finding “interesting” test data
Solutions: Test Case/Test Data Generation

A layered architecture of interacting generation algorithms

- Combinational solvers (OBDD, SAT) for small discrete data ranges and abstracted constraints
- Linear programming for simple, frequently occurring constraints of type \((x_1 < c_1) \land \ldots \land (x_n < c_n)\)
- Optimisation algorithms for real-valued constraints involving mathematical functions
- Solvers for differential equations
Solutions: Tool Chain

Integration of

- CASE tools for model development
- Solvers for mathematical problems: differential equations, constraint programming, optimisation
- Transformers to Intermediate Model Representation
- Code generators
- Test generators
- Simulation environment
- Hardware-in-the-loop test execution platform
Part1 – Tool chain - overview

Specification Front-Ends
- UML-Case Tool
- Matlab/Simulink
- State/Event Matrices
- other formalisms

UML2.0 Diagram Transformers

Intermediate Model
- Timed State Machines
- States/Events/Conditions/Actions
- Time-continuous Flows

Test Case Generator

Code/Test Data Generator

Distributed Test Program + Test Data

RT-Tester Tool Box

RT-Tester Engine – Hard Real-Time Test Execution Environment
Test case generation and test oracles for reactive real-time systems

Example: Signal Lamp Control

- System under test (SUT) switches relays on/off
- SUT checks feedback from relays
- SUT checks whether current flows through lamp filament
- SUT takes bouncing of discretes/relays into account
- SUT checks whether desired state is reached in time and is kept stable

Goal: Find test suite for checking SUT against Real-Time Statecharts specification
Example: Signal Lamp Control

1. Switch-on
2. Set timer for relais feedback
3. Set timer when to reach stable ON status
4. Set timer for current feedback
   - [f] / r(tF); s(tG)
   - [not(f)] / s(tF); r(tI)
   - [i] / r(tI)
   - [not(i)] / fail_current
   - [not(f and i)] / fail

Feedbacks:
- relais status
- current
- fail, fail_relais
- fail_current
- reset timer for current feedback

Controls:
- relais on/off
- set timer for relais feedback
- set timer when to reach stable ON status
- set timer for current feedback
- reset timer for current feedback

System Under Test

Jan Peleska 33
Example: Signal Lamp Control – Region Graph

1. \( t_F = 1 \) and \( t_G = 2 \)
2. \( 0 < t_F < 1 \) and \( t_F + 1 = t_G \)
3. \( t_I = 1 \) and \( 0 < t_G < 1 \)
4. \( \text{FAIL} \)

- \( [f] \)
- \( [\neg f] / \text{fail} \)
- \( / \text{fail} \)
- \( / \text{fail} \)
- \( / \text{fail} \)
Distinguishing Traces

- For each state in (minimal) region graph, determine I/O traces distinguishing this state from all other ones
- Example: distinguish states * and **:
  - Trace
    \[
    < (0, [f]), (\varepsilon, [\neg(f)]), (\varepsilon + 1, fail_{relais}) >
    \]
  - is possible for * and some \(0 < \varepsilon < 1\), but not for **.
  - Trace
    \[
    < (0, [f]), (\varepsilon, [\neg(f)]), (\delta, fail) >
    \]
  - is possible for ** and some \(0 < \varepsilon, \delta < 1\), but not for *.
Test Strategy

- For each specified state $s$, generate input trace which should force the system under test (SUT) into $s$ (deterministic behaviour required!)

- Apply inputs of all distinguishing traces of $s$ and check whether they lead to the specified outputs
Evaluation

Good news:
- Test strategy can prove correctness of real-time behaviour

Problems:
- Size of region graph is only bounded by $n! M^n$
  - $n$ number of timers
  - $M$ largest timer value
- Need to visit each state repeatedly for each of its distinguishing traces
Test data generation

- In general, test cases are specified on an abstraction of input, output and state data.
- Interface modules are suitable components of a test automation system to perform:
  - Refinement from abstract test case data to concrete interface data and driver calls.
  - Abstraction of concrete interface data to logical signals defined on test case level.
Conclusion

- Trustworthy algorithms for testing both combination and state-based real-time systems exist.
- For state-based systems, the number of test cases (i.e., traces to be checked) is too high to be executed in sufficient time.
Ongoing research – integration of algorithms into RT-Tester

- Test trace selection based on hazard analysis
- Test trace selection based on specification mutations