Software Testing IV

Prof. Dr. Holger Schlingloff
Humboldt-Universität zu Berlin
and
Fraunhofer Institute of Computer Architecture
and Software Technology FIRST
Outline of this Lecture Series

- **2006/ 11/ 24**: Introduction, Definitions, Examples
- **2006/ 11/ 25-1**: Functional testing
- **2006/ 11/ 25-2**: Structural testing
- **2006/ 11/ 26-1**: Model-based test generation
- **2006/ 11/ 26-2**: Specification-based test generation

- Next week: Your turn!
Outline of This Lecture

• Test generation from Finite State Machines
• Test generation from UML StateCharts
• Test generation from Timed Automata
Description of Systems

- Finite automata have been known since the 1960’s
  - Moore / Mealy: used to describe relations between words (input sequence is transformed into output sequence)
  - Rabin / Scott: used to describe sets of words (accepting / non-accepting states)

- Can be used to describe the control flow of any system
  - states are (equivalence classes of) configurations of the SUT
  - transitions are actions (external or internal) changing the state
Labelled Transition Systems

• “finite automaton without accepting states”
• formally: \((S, L, T, s_0)\)
  - \(S\): finite or countable nonempty set of states
  - \(L\): finite set of labels, \(\tau \notin L\)
  - transition relation \(T \subseteq S \times (L \cup \{\tau\}) \times S\)
  - \(s_0\) initial state

• Run=finite sequence \(\{(s_i, s_{i+1})_{i \in N}\}\), where \(s_0\) is the initial state and \((s_i, s_{i+1}) \in T\)

• Trace = sequence of observable actions (labels\(\neq \tau\)) of a run

• Undirected communication! Actions just “happen”!
Example: A Light Switch

- can be switched up and down
- may internally switch off
- cf. windshield wiper example
Test Generation

Remarks:
- each “box” can be manual or automated
- if everything is automated, only the tools are tested

Requirements:
- SUT must accept inputs from test driver
- SUT must provide recognisable outputs for test driver
- SUT must be resettable by test driver
- SUT must be deterministic
Conformance

• Given an LTS, what does it mean that an implementation is “correct” with respect to this model?
  - same structure? same states? same state classes?
  - same behaviour? same observable behaviour?
  - same choices? fewer choices? more choices?
  - same timing? more specific timing?

• Fault model (extra/missing state, unexpected output, quiescence, …)

• Conformance notions (Tretmans)
Trace Refinement

- Observable behaviour: set of sequences of visible actions of the SUT
- Traces (P) = set of observable behaviours of process P
- $\text{Imp} \leq_T \text{Spec}$ gdw. $\text{Traces(}\text{Imp}) \subseteq \text{Traces(}\text{Spec})$

pre-order (transitive, reflexive)

$\Rightarrow$ minimal element is the empty trace

$\Rightarrow$ comparable to language inclusion in finite automata theory
Testing Trace Refinement

• Test case = trace
• Test suite = set of traces
• Test execution of trace $\sigma$ for Imp and Spec:
  • $\sigma \notin \text{Traces}(\text{Imp})$ $\Rightarrow$ pass
  • $\sigma \in \text{Traces}(\text{Imp}) \cap \text{Traces}(\text{Spec})$ $\Rightarrow$ pass
  • fail, else
• Verdict of a test suite is the conjunction of individual verdicts

• Complete test suite: set of all traces over the alphabet
• $\text{Imp} \leq_T \text{Spec}$ iff complete test suite passes
• not feasible, thus additional hypotheses (length of traces, number of certain actions etc.)
Failures

- Failure = (σ, A), where σ is a sequence of observable actions, A is a set of actions.

- Failures(P) = set of failures (σ, A), such that there is an execution of P where σ can be observed, and afterwards no action from A is activated (P after σ refuses A).

- in automata: „non-transitions“

- $Imp \leq_F Spec$ iff Failures($Imp$) ⊆ Failures($Spec$)

- also a partial order

- finer than trace-Refinement:

  $Imp \leq_F Spec \Rightarrow Imp \leq_T Spec$
Failure Refinement

- $Imp \leq_F Spec$ iff $\text{Failures}(Imp) \subseteq \text{Failures}(Spec)$
- $Imp \leq_F Spec$ iff $(Imp\text{ after }\sigma\text{ refuses }A)$ implies $(Spec\text{ after }\sigma\text{ refuses }A)$
  - $Imp$ may only refuse those actions which are also refused by $Spec$
  - $Imp$ may only perform those actions which are allowed by $Spec$
  - $Imp$ has „less deadlocks“ than $Spec$
- Refinement with respect to this relation
  - transformational development
  - correctness proofs
Testing of Failure Refinement

- Test suite $T = \text{set of failures } (\sigma, A)$
- Complete test suite = set of all failures for a set of observable events
- Verdict of a test $(\sigma, A)$ with respect to $\text{Imp}$ and $Spec$
  - $\sigma \notin \text{Traces(Imp)} \Rightarrow \text{pass}$
  - $(\sigma, a) \in \text{Traces(Imp)}$ for some $a \in A \Rightarrow \text{pass}$
  - $(\sigma, A) \in \text{Failures(Imp)} \cap \text{Failures(Spec)} \Rightarrow \text{pass}$
  - fail, else
- Verdict of a test suite: all test cases pass
- Test of an implementation $\text{Imp}$
  - $\text{Imp} \leq_{F} \text{Spec}$ iff complete test suite passes
  (under certain side-conditions)
Conformance

• $Imp \text{ conf } Spec$ iff for all $\sigma$ in Traces($Spec$):
  $\left( Imp \text{ after } \sigma \text{ refuses } A \right) \rightarrow \left( Spec \text{ after } \sigma \text{ refuses } A \right)$
  - for action sequences of the specification same as $\leq_F$
  - $Imp$ may implement „additional functionality“
  - weaker than $\leq_F$ (i.e. $Imp \leq_F Spec \rightarrow Imp \text{ conf } Spec$)

• conformance testing similar as with failure-refinement-testing

• widely used as a correctness criterion
IOCO

- Taking also inputs and outputs into consideration
- All inputs are always enabled
- \( \text{out}(P \text{ after } \sigma) = \{a! \mid P \text{ may execute } \sigma \text{ and then output } a!\} \)
- \( \text{Imp ioco Spec iff for all } \sigma \text{ in Traces}(Spec): \)
  \( \text{out}(\text{Imp after } \sigma) \subseteq \text{out}(Spec \text{ after } \sigma) \)
- Idea
  - \( \text{Imp} \) is more deterministic than \( \text{Spec} \) with respect to specified inputs
  - \( \text{Imp} \) may implement additional functionality for unspecified inputs
Implementation: TGV

- TGV “Test Generation with Verification”
- Conformance testing for reactive systems, black box test
- Automated test generation from LTS, IOCO
- Interaction via PCOs
- Verdict
  - fail: a non-conformance was observed
  - pass: trace could be executed in the SUT
  - inconc: else
TGV Testing Purposes

• A testing purpose in TGV is a “small” LTS with additional transitions ACCEPT, REFUSE
• TGV builds the cartesian product of spec and testing purpose
• Test generation: determinisation of LTS and TP, enumeration of traces
UML StateCharts

- can be seen as LTS with
  - hierarchy
  - parallelism
  - inheritance
Parallelism

• What is the meaning of parallelism in the specification?
  ▪ structural: must be implemented in parallel
  ▪ engineering: may be implemented in any order
  ▪ pragmatic: will be implemented according to the tool’s scheduling strategy
Test Generation from StateCharts

- ATG ("automated test generator"): Add-on to the UML-tool Rhapsody by ILogix / IBM

- First, the model is translated into C++ by the Rhapsody code generator
- Then, inputs and outputs to the model / SUT are identified
- Then, ATG constructs test cases from the generated code according to certain coverage goals
  - all states
  - all transitions
  - MC/DC
A “Real-Life” Example

- A safety protocol for industrial automation
Real Time

- Real Time concepts in UML (-state diagrams)
  (a) after (time) as a trigger
  (b) absolute time point (after start) as a trigger

- Informal semantics
  (a) transition will be executed \( t \) time units after becoming active
  (b) transition will be executed at the given time point

- Often, this is not sufficient
  - no minimal / maximal waiting times
  - no possibility of using several clocks
Timed Automata

• Extension of LTS by clocks
  ▪ All clocks are constantly running (no stopwatches)
  ▪ All clocks run at the same speed (perfect clocks, $t'=1$)
  ▪ Clocks can be reset by transitions
  ▪ Clocks can influence the switching of transitions

- One clock $x$.
- No invariant at $s_1$, so the system may stay arbitrarily long in $s_1$.
- When transitioning to $s_2$ by $a$ the clock will be reset to 0.
- In $s_2$ the clock is running.
- After at least 1 time unit the transition to $s$ is possible, after at most 2 time units it must happen.
Example

- Double-click switch
  - Click on, click off
  - After clicking twice fastly in a row, it becomes brighter

- Additional requirement
  - after at most 300 ms turn darker again

More about timed automata: Rajeev Alur, Tom Henzinger
http://www.cis.upenn.edu/~alur/Talks/sfm-rt-04.ppt
Formal Definition

• Assume a set $X$ of time variables. A *timed condition* is a Boolean combination of formulas of the kind $x < c$, $x \leq c$ (where $c$ is any rational number).

• A timed automaton is a tuple consisting of
  - a finite set $L$ of *locations*
  - a subset $L_0$ of *initial locations*
  - a finite alphabet $\Sigma$ of *events*
  - a finite set $\Xi$ of *clocks* (clock-variables)
  - An *invariant* $\text{Inv}(s)$ for every location (timed condition, optional)
  - a finite set $E$ of *transitions* consisting of
    - source, target
    - event from the alphabet (optional)
    - timed condition (optional)
    - set of clocks to be reset (optional)
Semantics of Timed Automata

• Each timed automaton is assigned an infinite LTS:
  - states: \((l, v)\) where \(l\) is a location and \(v\) is an assignment of clocks with real numbers which satisfies \(\text{Inv}(l)\)
  - initial state: \((l_0, (0, \ldots, 0))\)
  - transitions
    - control transition: \((l, v) \xrightarrow{a} (l', v')\) if a transition \((l, a, g, r, l')\) exists such that \(v\) satisfies \(g\) and \(v' = v[r := 0]\)
    - time transition: \((l, v) \xrightarrow{d} (l', v')\) if \(l' = l\) and \(v' = v + d\) and both \(v\) and \(v'\) satisfy \(\text{Inv}(l)\)

• Each path through the transition system is an execution of the timed automaton
  - control and time transitions strictly alternating
  - semantics = set of (infinite) executions, non-Zeno
Determinism

• Attention: if e.g. the conditions are inconsistent, the set of executions may be empty

• Additional requirements to take care of such situations
  ▪ the timed conditions are mutually inconsistent (at least those for the same event)
  ▪ they sum up to true, i.e. the disjunction of all timed conditions is a tautology

• Often combined with other determinism requirements, e.g. input enabledness
Extensions of Timed Automata

- **Input / Output – Timed Automata**
  - partitioning of $\Sigma$ in inputs (i?), outputs (o!), and internal events
  - each transition can be labelled by an input or an internal event, and at the same time by several outputs

- **Additional variables $v_1, \ldots, v_n$ with finite domains $W_1, \ldots, W_n$**
  - location = (place, values $(w_1, \ldots, w_n)$)

- **Parallel and hierarchical automata**
  - similar to StateCharts
  - usual automata product semantics (interleaving)
  - only for convenience in modelling
Test Generation from TA

- Which one to choose?
  (Random choice might result in „bad coverage“)
- Wanted: a strategy which covers all behaviours
Idea: Equivalence Classes

- Partition the infinite state space into finitely many "regions" such that all regions are behaviorally "similar"
- Cover the region graph with untimed methods
Quotients of Automata

• Recall the definition of the quotient automaton of a finite automaton with respect to a partitioning of the state space

  ▪ abstraction of certain variables
    - smaller domains
      e.g. \( \text{Int} \rightarrow \{-\text{maxint}, \ldots, -1\}, \{0\}, \{1, \ldots, \text{maxint}\} \)
    - elimination of variables (unit domain)

  ▪ states in the quotient automaton are equivalence classes of states in the original automaton

  ▪ this is a coarsening of the specification, the set of executions becomes larger
Quotients of Timed Automata

• With timed automata
  ▪ abstraction of clock variables: the language of the untimed automaton strictly encompasses the language of the respective timed automaton
  ▪ maybe too coarse (why did we introduce time after all?)

  ![Diagram showing two states: one with a transition labeled 'x:=0' and another with 'x>10'.]

• Two quotient constructions have been proposed
  ▪ Region equivalence
  ▪ Zone equivalence
Regions

- Finite partitioning of the state space

**Definition region equivalence:**
Let $c_{\text{max}}$ be the largest constant occurring in the automaton.

$v \cong_R u$ iff
- the integral part of all clock valuations is equal or both $>c_{\text{max}}$.
- the fractional part is both $=0$ or both $>0$.
- if $x < c_{\text{max}}$ and $y < c_{\text{max}}$ are clocks, then $(x \leq y \text{ in } v \iff x \leq y \text{ in } u)$

A region (equivalence class)
Zones

- Finite set of inequalities
- All possible <-relations between all clock variables

Definition:

\[ w \cong_Z w' \text{ iff } \]

\[ \begin{align*} w \text{ and } w' & \text{ satisfy the same inequalities} \\
    x_i < c, \ x_i = c, \ x_i - x_j < c, \ x_i - x_j = c \end{align*} \]

where \( x_i, x_j \) are clocks and \( c \leq c_{\text{max}} \) is some constant

- It is sufficient to check this condition for linear combinations of variables (\( c = k_1 c_1 + \ldots + k_n c_n \))
Finiteness of the State Space

• The relations $\cong$ have a finite index
  - i.e. there are only finitely many reachable regions / zones
  - i.e. the region/zone graph is finite

• Proof idea (regions)
  - follows from the definition of $\cong_\mathbb{R}$: there are only finitely many integral parts $\leq c_{\text{max}}$ and finitely many comparisons of fractional parts

• Proof idea (zones)
  - with finitely many rational numbers there are only finitely many linear combinations smaller than a given bound
  - each timed automaton contains only finitely many constants
  - thus there are only finitely many conditions of the mentioned form
Construction of the Region Graph

- Theorem: the region graph satisfies the same safety properties as the original timed automaton.
  - Thus, a complete test suite for the region graph will uncover all safety errors in the timed automaton.
Example

• #Regions depends on #Locations, #Constants, #Clocks
• i.A. exponential in the number of clocks and the number and size of constants (PSPACE-complete)
UppAal, Kronos, Rabbit...

- Tools for the construction of the region graph
  - animated simulation
  - temporal verification
  - test generation (UppAal)

- different internal representations
  - sets of inequalities
  - binary decision diagrams (BDDs)
  - difference bound matrices (DBMs)

--- Presentation UPPAAL? ---
