Model-Based Testing

(DIT848 / DAT260)

Spring 2014

Lecture 12

FSMs, EFSMs and ModelJUnit

Gerardo Schneider
Department of Computer Science and Engineering
Chalmers | University of Gothenburg

Outline

- The Qui-Donc example
- Modeling Qui-Donc with an FSM
- Some simple techniques on how to generate tests from the Qui-Donc model
- EFSM
- The ModelJUnit library
- A Java "implementation" of an EFSM for the Qui-Donc example
- Remark: **No test automation today!**

Qui-Donc

- France Telecom service to get name and address given a phone number (vocal service)
- Informal requirements of the system in what follows

Qui-Donc: Informal requirements (1)

Utting & Legeard book:
Sec 5.1.1 pp.140!

Source: M. Utting and B. Legeard, *Practical Model-Based Testing*

Qui-Donc: Informal requirements (2)

Utting & Legeard book:
Sec 5.1.1 pp.141!

Modeling Qui-Donc with FSM

- Decision: What to abstract?
  - Too big! (FSM cannot represent data structures, variables, timeouts, etc.)

What would you abstract?

Suggest some interesting cases to keep (representative), others that might be "forgotten"

Groups 2-5 persons: 5-10 min

Modeling Qui-Donc with FSM

- Decision: What to abstract?
  - Too big! (FSM cannot represent data structures, variables, timeouts, etc.)

- For testing purpose our abstraction considers:
  - The 4 "special" keys (1, 2, *, #)
  - 4 representative numbers
    - 18 - Emergency number
    - num1 (03 81 11 11 11) – disconnected number (not in the database)
    - num2 (03 81 22 22 22) – we know address and name
    - bad (12 34 56 78 9) – wrong number (9 digits instead of 10)

Modeling Qui-Donc with FSM
Relating Inputs with the Real World

- **Input alphabet** of our model: {dial, num1, num2, bad, 18, 1, 2, *, #, wait}
- dial: pick up phone, dial Q-D service, wait for response
- 1, 2, *, #: press the corresponding key
- 18: press 1 then 8, then # (within 6 sec)
- num1: press all digits followed by # (within 20 sec)
- num2 (bad): press all digits followed by # (as quick as possible)
- wait: wait without pressing anything until Q-D does somehting (timeout: 20 sec for ENTER state, 6 sec for others)

Qui-Donc FSM Model Outputs

Example of Input/Output sequence:

dial/WELCOME,
wait/WELCOME,
*/ENTER,
num1/NAME+INFO,
2/ADDR,
wait/INFO,
wait/BYE

Utting & Legeard book: Table 5.1 pp.146!

Source: M. Utting and B. Legeard, *Practical Model-Based Testing*

Modeling Qui-Donc with FSM

- We will use a special kind of FSM
- A Mealy machine is an FSM where
  - Each transition is labeled with input/output (exactly one input per transition; output may be empty)
  - Must have one initial state
  - May have one or more final states
- Generated tests should start in inital state and finish in one of the final states
  - If no final state: allowed to end in any state

Qui-Donc FSM Model

- Not easy to model timeouts in FSMs
- To model them we have 3 different states Star1, Star2, Star3, (similarly for Enter and Info)
- That's why we have repeated wait/_ on the transitions from those states (message repeated up to 3 times)

Utting & Legeard book: Fig. 5.1 pp.145!

Source: M. Utting and B. Legeard, *Practical Model-Based Testing*

Representations of FSM
State Table

Utting & Legeard book:
Table 5.2 pp.147!

Source: M. Utting and B. Legeard, *Practical Model-Based Testing*

"Properties" of FSM

- Deterministic
  - For every state, every outgoing transition labeled with different input
- Initially connected
  - Every state reachable from initial state
- Complete
  - For each state, outgoing transitions cover all inputs
- Minimal
  - No redundant states (no 2 states generating the same set of input/output sequences with same target states)
- Strongly connected
  - Every state is reachable from every other state

Generating Tests
(from the Qui-Donc model)

We will see in what follows:

- State, input, and output coverage
- Transition coverage
- Explicit test case specifications

- Complete testing methods
  - More powerful FSM test generation

Generating Tests: State, input, and output coverage

- State coverage: Percentage of FSM states visited
  - Q-D: 1 test, 12 transitions 100% (dial,wait,wait,*,wait,wait, 18,*,num2,wait,wait,wait – omitting outputs)
  - State coverage in FSM similar to statement coverage in PL
- Input coverage: Nr. of diff. input symbols sent to SUT
  - Q-D: 1 test, 90% out of 10 inputs (dial/WELCOME, */ENTER, bad/ERROR, num1/SORRY, num2/NAME, 1/SPELL, 2/ADDR, */ENTER, 18/FIRE, wait/BYE)
- Output coverage: Nr. of diff. output responses from SUT
  - Q-D: same test sequence as for *Input coverage*, covers 9/11 outputs

Generating Tests: Transition coverage

- How many FSM transitions have been tested
- Random path: will eventually cover all
- Transition tour: best way – in particular the Chinese Postman algorithm (CPA)
  - CPA finds the shortest path
- Transition coverage in FSM similar to branch coverage in PL
- **Full transition coverage is a good minimum to aim!**
- See Utting&Legeard, listing 5.2 (pp.152) for the output of the Chinese Postman algorithm in Qui-Donc

Generating Tests: Explicit test case specifications

- Useful to write an explicit test case specification
  - Define which kind of test to be generated from the model (low-level)
  - High-level test designed by **engineer**; low-level details and expected SUT output from the **model**
  - Q-D (example) - Test slow people failing to complete input before timeout: *,Star3,*,Enter3,*,Info3,*
    - Regular expression over seq of **states**
    - "*" is a wildcard (any seq of actions)
    - Shortest test case satisfying the above: dial/WELCOME,wait/WELCOME, wait/WELCOME, */ENTER,wait/ENTER,wait/ENTER,num2/NAME,wait/INFO,wait/INFO,wait/BYE

Generating Tests: Complete testing methods

- Many complete test generation methods for FSMs were invented (60's-80's): D-method, W-method, Wp-method, U-method, etc
  - Guarantees that SUT is "equivalent" to the FSM
  - Strong assumptions on the FSM: deterministic, minimal, complete, strongly connected, **and** must have the same complexity of the SUT
  - Some relaxation possible: weaker results

Read Utting&Legeard section 5.1.4 (pp 155-157), and references therein

Extended FSM (EFSM)

- EFSMs are like FSMs but more expressive (internal variables encode more detailed state information)
  - In FSM: Many $\mathbf{Enter}_i$ states
    In EFSM: one **Enter** state + **timeouts** variable to count nr of timeouts
- It seems to have a small nr. of **visible states**: in reality a much larger nr. of **internal states**!
- Mapping large set of internal states of an EFSM into the smaller set of visible states: abstraction

Extended FSM (EFSM)

"An EFSM can model an SUT more accurately than an FSM, and its visible states define a 2nd layer of abstraction (an FSM) that drives test generation"

Source: M. Utting and B. Legeard, *Practical Model-Based Testing*

The two levels of abstractions give better control - used for different purpose:

- Medium-size state space of EFSM (and code in transitions) used to model the SUT behavior more accurately and thus generate more precise inputs and oracles for the SUT
- Smaller nr. of visible states of EFSM: defines an FSM used to drive test generation (eg, algorithm for transition tour)

Extended FSM (EFSM)
Example

- Assume an SUT with infinite state space (integers)
- Model as EFSM with 2 int var (x,y: 0..9)
  - 10x10=100 internal states
- Partition state space into 3 (based on our test objectives):
  A (y>=x), **B** (y<x and x<5),
  C (y<x and x>=5)
- Code in transitions to make state updates
  - AB1: x,y := 1,0 (no guard)
  - AB2: y := 0 (guard: [x<5])
  - AB3: y := y-1
    (guard [x=y and 0<x<5])

Utting & Legeard book: Fig. 5.2 pp.158

Source: M. Utting and B. Legeard, *Practical Model-Based Testing*

The ModelJUnit Library

- A set of Java classes designed as an extension of JUnit for MBT
- Allows (E)FSM to be written in Java, and tests are run as for JUnit
- Provides a collection of traversal algorithms for generating tests from the models
- Usually used for online testing (tests executed while being generated)
- EFSM plays 2 roles
  - Defines possible states and transitions to be tested
  - Acts as the adaptor connecting model and SUT (more on this in next lecture)

The ModelJUnit Library

- Each EFSM must have at least the following methods
- Object getState()
  - Returns the current visible state of EFSM (defines an abstraction function between EFSM internal state to EFSM visible states)
- Void reset(boolean)
  - Resets the EFSM to initial state – When online testing, also reset SUT (or create new instance)
- @Action void $name_i$()
  - Define transitions of the EFSM (also sends test inputs to SUT and check answers)
- boolean $name_i$Guard()
  - Guard of the action method; actions with no guard defined have an implicit true guard

Qui-Donc's EFSM
(In Java)

states

Initial state

Get current state

Reset

Utting & Legeard book: List. 5.3 pp.163

Source: M. Utting and B. Legeard, *Practical Model-Based Testing*

Qui-Donc's EFSM

(In Java)

3 transitions labelled with "star" (*), from states "Star", "Emerg", and" "Info"

Guard of "star"

Input (action) "star"

Transitions with input "star" (*) incoming to "Enter" state

Utting & Legeard book: List. 5.4 pp.164

Source: M. Utting and B. Legeard, *Practical Model-Based Testing*

EFSM of Qui-Donc (from the Java model)

Utting & Legeard book:
Table 5.2 pp.147!

Source: M. Utting and B. Legeard, *Practical Model-Based Testing*

Group exercise

- Is the graph an Euler graph?

No!

- Eulerize it!

Add a "copy" of **num18**

- Give (abstract) test cases to obtain 100% transition coverage

Proposed solution:

wait, dial, wait, star, num1, bad, wait, num2, key1, key2, wait, star, num18, star, num18, wait

Groups 2-5 persons: 5-7 min

Validating the Model

- Possible to write a main method to call methods iteratively
- Do a manual traversal using transition tour (e.g. Chinese Postman)
- You might find errors in your model
  - Correct, iterate

Generating Tests from the Model

- In the Qui-Donc - You can generate a random walk to get a test sequence randomly generated
- You can use the output as a manual test script
- To manually test the real system by giving the inputs and checking the expected output

Final Remarks

- We have used ModelJUnit to generate offline testing only
  - The Qui-Donc example is a physical device and we used EFSM and ModelJUnit to automatically generate test sequences to be manually tried on the physical device
- For online testing you need to define an adaptor, which links the model to the SUT
  - This is possible in ModelJUnit (next lecture)

References

- M. Utting and B. Legeard, *Practical Model-Based Testing*. Elsevier - Morgan Kaufmann Publishers, 2007
  - Chapter 5 (Sections 5.1-5.2)