Input Data Generation for Model-based Testing

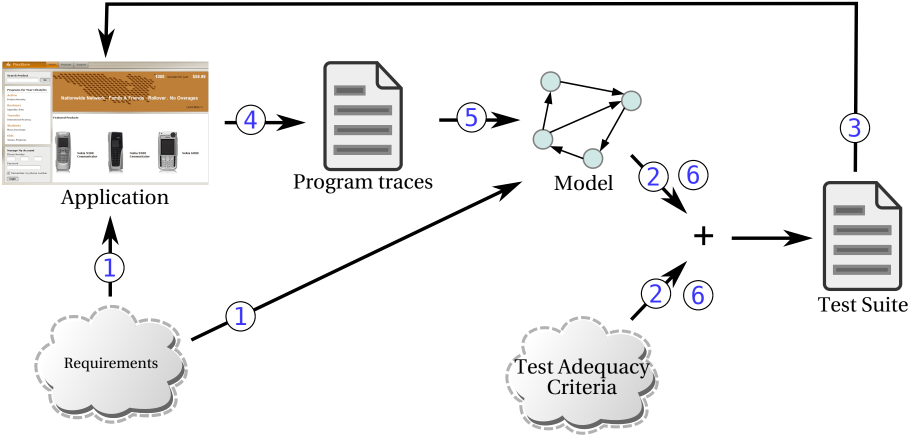
R. Tiella (tiella@fbk.eu) and P. Tonella (tonella@fbk.eu)
Software Engineering Unit
FBK - Trento, Italy

3rd **FITTEST Industrial Day - May 31, 2013**

Outline

- Background
  - Model-based Testing
  - Test Adequacy Criteria
- Proposed Approach
  - Working example
  - Input Data Generation Problem
  - Intuition behind the solution
  - the Tool
  - a Case study
- Conclusions

Model-based Testing - the Process



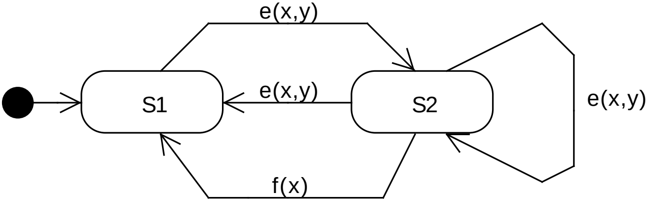
Model-based Testing - the Model

Given that:

- SUT' state is observable by "inspector" messages
- SUT' state changes due to "mutator" messages
- observable states are abstracted collapsing "equivalent" ones

SUT is modeled by a Finite State Machine (FSM):

- with parametrized events, e.g. $e(x, y)$
  - method calls, http requests, etc.
- non-deterministic



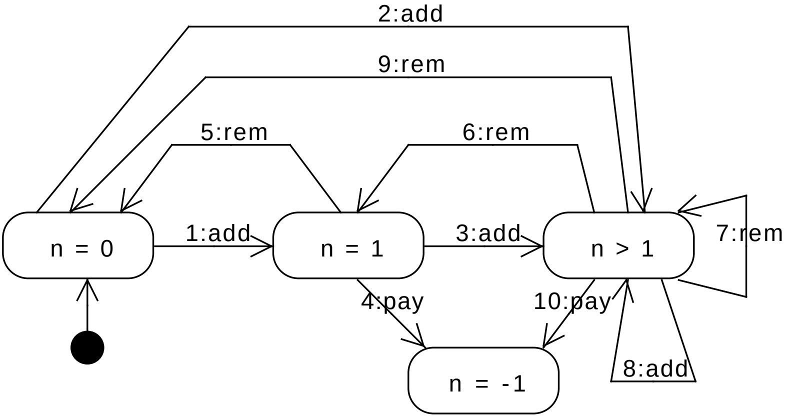
Test Adequacy Criteria

- "good" test cases:
  - can hardly be defined a-priori
  - informally are effective, cheap, helpful to identify the underlying fault
- test adequacy criteria:
  - are a means to concretely specify the extent to which a test suite has to exercise a SUT
- currently, well-accepted criteria are:
  - **all transitions coverage**: [fsm] every transition in a FSM has to be traversed by at least a test case
  - **all branches coverage**: [source code] every branch in a program's CFG has to be executed by at least a test case

Subject Under Test Example

```
package cart;

public class Cart {

    public Cart() { ... }

    public void add(int c) { ... }

    public void rem(int c) { ... }

    public void pay() { ... }

    public int n() { ... }

}
```

SUT Model



- actual Cart's state is abstracted with 4 states
- transitions are triggered by events: `add(int c)`, `rem(int c)`, `pay()`

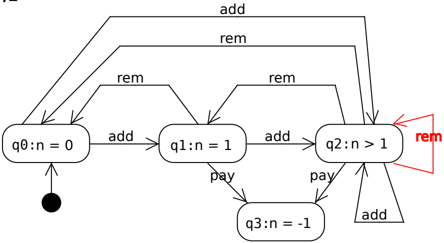
FSM specification

```
mutators {
    a := add(int);
    r := rem(int);
    p := pay();
}

inspectors {
    int n:=n();
}

states {
    n0 [initial] { n == 0; } ;
    n1 { n == 1; } ;
    n2 { n > 1; } ;
    n3 { n == -1; } ;
}
```

```
transitions {
    n0 -> n1 { a; } ;
    n0 -> n2 { a; } ;

    n1 -> n0 { r; } ;
    n1 -> n2 { a; } ;
    n1 -> n3 { p; } ;

    n2 -> n2 { a; } ;
    n2 -> n2 { r; } ;
    n2 -> n3 { p; } ;
    n2 -> n1 { r; } ;
    n2 -> n0 { r; } ;
}
```

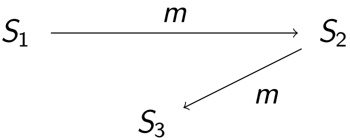
Input Data Generation Problem (Example)

- Test Requirement:
  - traverse $rem(n) : q_2 \longrightarrow q_2$
- Problem:
  - devise a path that reaches state q_2
  - find out the proper inputs:
    - to reach q_2 and then
    - to traverse $rem : q_2 \longrightarrow q_2$



Intuition behind the proposed approach

Observation: if the SUT logic was implemented following a typical FSM design pattern ...

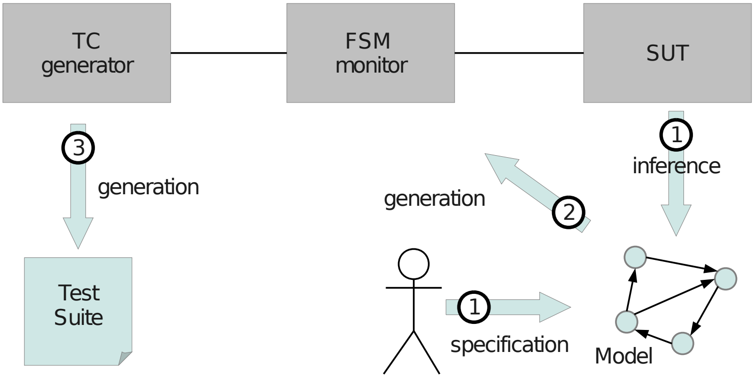



```
// pseudo-code to handle
// transitions triggered by m

if (s==1)
    s=2 // S_1 -> S_2
elseif (s==2)
    s=3 // S_2 -> S_3
else
  throw unexpected
```

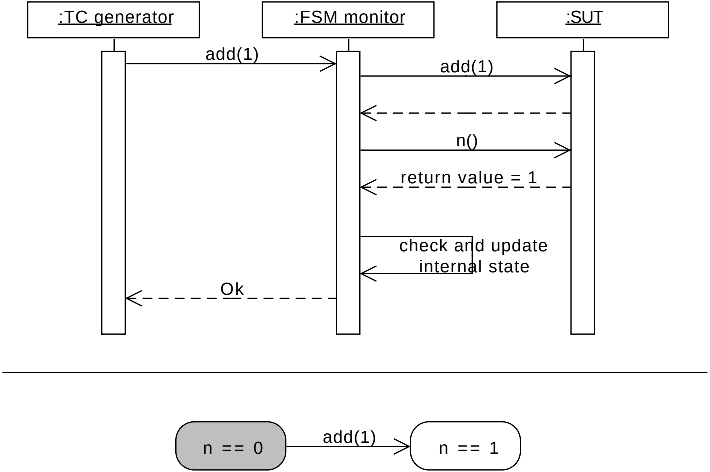
... CFG branches corresponds to FSM transitions.

FSM Monitor Generation

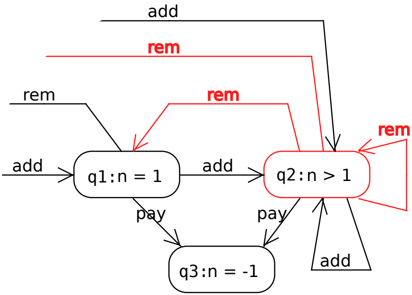
- Introduce a component which encodes the FSM logic in its branches, a sort of “man-in-the-middle”, so that exploring its branches means exploring SUT transitions.



FSM Monitor Generation



FSM Monitor Generation




```
void rem(int n) {
  sut.rem(n);
  switch (s_state) {

  case 2:
    if ( n > 1 ) {
      s_state = 2;
    } else if ( n == 1 ) {
      s_state = 1;
    } else if ( n == 0 ) {
      s_state = 0;
    } else {
      throw new UnderSpecEx(...);
    }
    break;

  case 1:
    ...
}
```

FSM Monitor Generation

- **observation:** From the FSM, we can build a class A so that:

if the SUT is compliant with FSM, then

All Transitions Criterion for FSM is satisfied
iff
All $\overset{NonError}{\curlyvee}$ Branches Criterion for A is satisfied

- **theoretical result**: it's true!
- **COTS are available**: we can use an already available TC generator for branch coverage (e.g. **Evosuite**)
- **generator implementation**: template-based ⇒
  - highly configurable
  - different implementation styles can be supported
  - graphical representations can be produced with proper templates

TC Generator - Evosuite

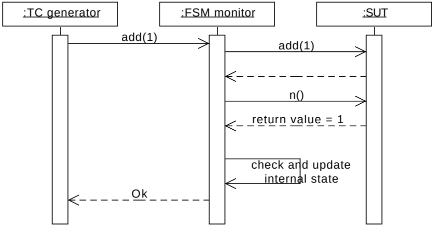
Evosuite[1] is a search-based test suite generator for Java

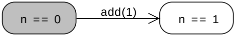
- based on an evolutionary approach, i.e. it mimics natural evolution to optimize branch coverage:
  - candidate solutions in the search space are modeled by chromosomes
  - good candidate solutions have high branch coverage level
  - chromosomes of "best" individuals are combined by cross-over
  - some chromosomes are mutated to maintain diversity and to introduce new alleles
- uses whole test suite generation strategy:
  each chromosome encodes a whole test suite

[1] http://www.evosuite.org/ - G. Fraser and A. Arcuri

Detecting Diverging Behavior - 1

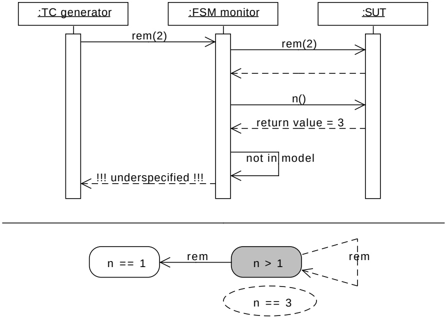
SUT	Model	Out	Effect
Ok, q'	$q \xrightarrow{m} q'$	Ok	invocation accepted
Ok, q'	$q' \notin T(q, m)$	Undersp. Exc.	TC discarded
Error	$m \in Out(q)$	Infeasible Exc.	TC discarded
-	$m \notin Out(q)$	Infeasible Exc.	TC discarded





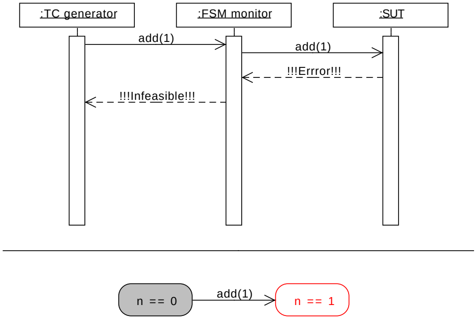
Detecting Diverging Behavior - 2

SUT	**Model**	**Out**	**Effect**
Ok, q'	$q \xrightarrow{m} q'$	Ok	invocation accepted
Ok, q'	$q' \notin T(q, m)$	Undersp. Exc.	TC discarded
Error	$m \in Out(q)$	Infeasible Exc.	TC discarded
-	$m \notin Out(q)$	Infeasible Exc.	TC discarded



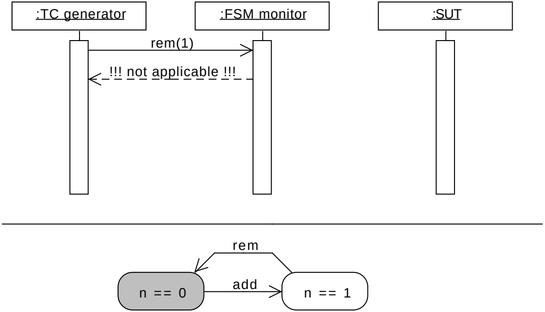
Detecting Diverging Behavior - 3

SUT	Model	Out	Effect
Ok, q'	$q \xrightarrow{m} q'$	Ok	invocation accepted
Ok, q'	$q' \notin T(q, m)$	Undersp. Exc.	TC discarded
Error	$m \in Out(q)$	Infeasible Exc.	TC discarded
-	$m \notin Out(q)$	Infeasible Exc.	TC discarded



Detecting Diverging Behavior - 4

SUT	Model	Out	Effect
Ok, q'	$q \xrightarrow{m} q'$	Ok	invocation accepted
Ok, q'	$q' \notin T(q, m)$	Undersp. Exc.	TC discarded
Error	$m \in Out(q)$	Infeasible Exc.	TC discarded
-	$m \notin Out(q)$	Infeasible Exc.	TC discarded



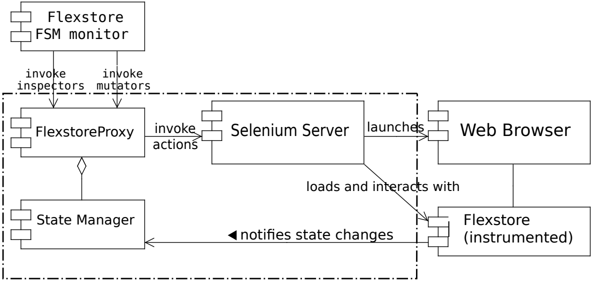
Test Suite

```
//Test case number: 0
   testCart0.add(1);
   testCart0.add(1);
   testCart0.add(1);

//Test case number: 1
   testCart0.add(2);
   testCart0.pay();

//Test case number: 2
   testCart0.add(11);
   testCart0.rem(11);

//Test case number: 3
   testCart0.add(1);
   testCart0.rem(1);

//Test case number: 4
   testCart0.add(1734);
   testCart0.rem(1);

//Test case number: 5
   testCart0.add(2);
   testCart0.rem(1);

//Test case number: 6
   testCart0.add(1);
   testCart0.pay();
```

- Note: test cases are small thanks to Evosuite's minimization algorithm.

Flexstore

Search Product
Go
Programs for Your Lifestyles
Active
Product Warranty
Business
Upgrades, Data
Traveler
International Roaming
Students
Music Downloads
Kids
Manage My Account
Phone Number
Password
Remember my phone number
Login
1000 minutes for just $59.99
Nationwide Network . Family & Friends . Rollover . No Overages
Learn More >>
Featured Products
Nokia 9300 Communicator
Nokia 9500 Communicator
Nokia 6800



Flexstore FSM monitor
invoke inspectors
invoke mutators
FlexstoreProxy
invoke actions
Selenium Server
launches
Web Browser
loads and interacts with
State Manager
notifies state changes
Flexstore (instrumented)

Conclusions

- a method to resolve the input data generation problem was proposed
- it is base on the idea that "all transition coverage" criterion can be transformed into "all branches coverage" criterion for which already exists good solutions (e.g. Evosuite)
- the transformation was formally proved correct (not shown)
- the method was successfully applied to a real application

That's all

Thanks for your kindly attention,
any question?