

# Model-Based Testing

(DIT848 / DAT260)

Spring 2013

## Lecture 10 EFSMs and Executable Tests (in ModelJUnit)

Gerardo Schneider

Department of Computer Science and Engineering  
Chalmers | University of Gothenburg

# Summary of previous (2) lectures

- 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
  - Offline testing (not executable)
- 2 interactive exercises on EFSMs

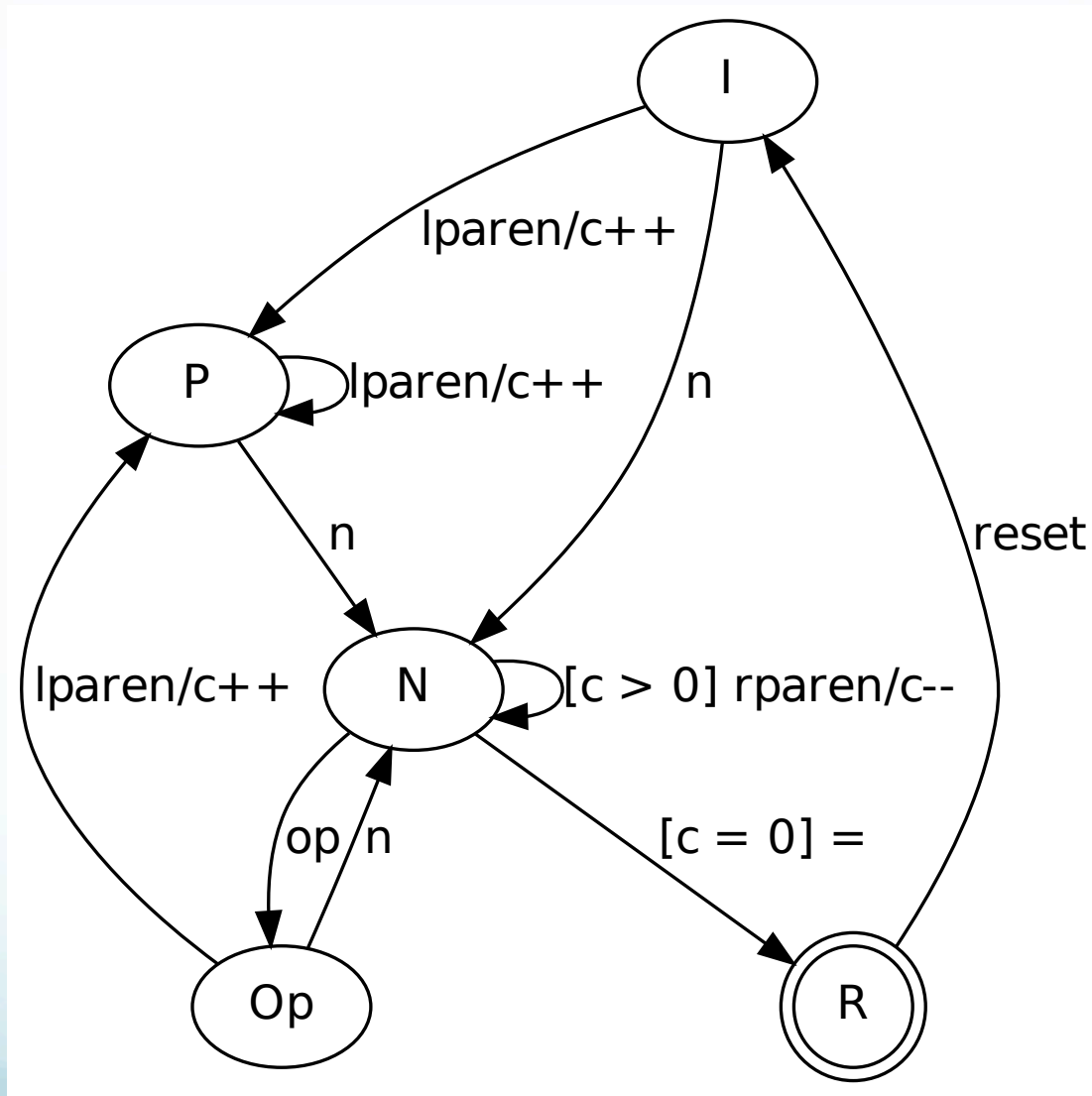
# Outline

- More interactive exercises on building an EFSM
  - Partial solution to the 1st part of Assignment 4
- Executable tests
- Online testing with ModelJUnit

# EFSM for Calculator (v.1)

- Write an EFSM for a calculator accepting (positive) integers, different operators ( $*$ ,  $+$ ,  $-$ ,  $/$ ), a reset operation, and parenthesis
- Assume numbers are full integers (not a string of digits)
- Assume that there is no need to check for division by zero
- The result is given when entering "=" (no need to "calculate" the result)
- After pressing "=" the result should be given and the calculator is reset
  - I.e., it is not possible enter an expression "1+2=+4" and expect to get 7 as result (computing 1+2 first and adding 4 to the result)
- For this first version: Assume that inputs with only one operator between two operands is accepted (i.e. something like "1+\*2" is not accepted)

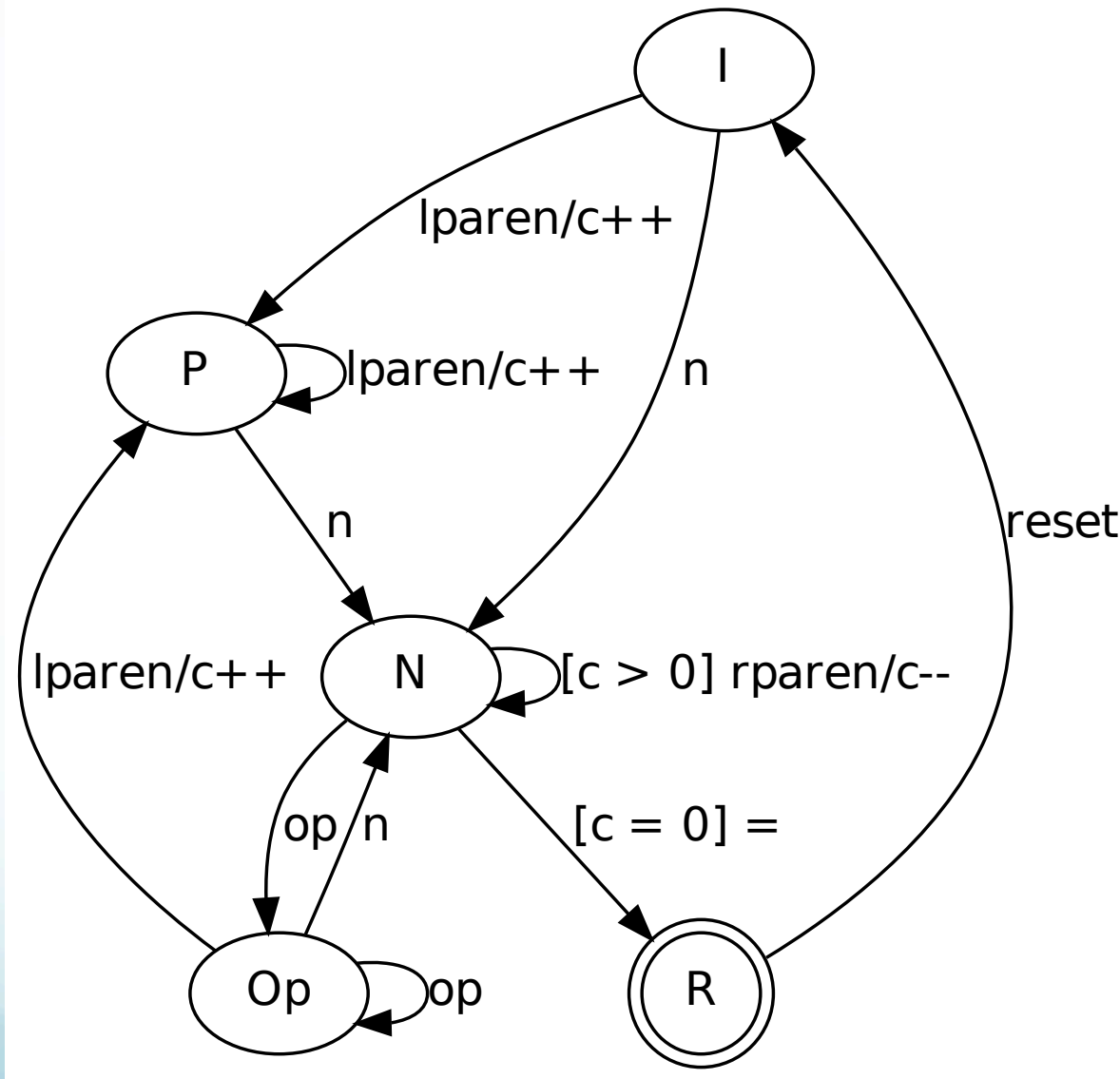
# EFSM for Calculator (v.1)



# EFSM for Calculator (v.2)

- Modify the previous EFSM to allow any number of operators between two operands
- The last operator is the one being considered, all the others being discarded

# EFSM for Calculator (v.2)

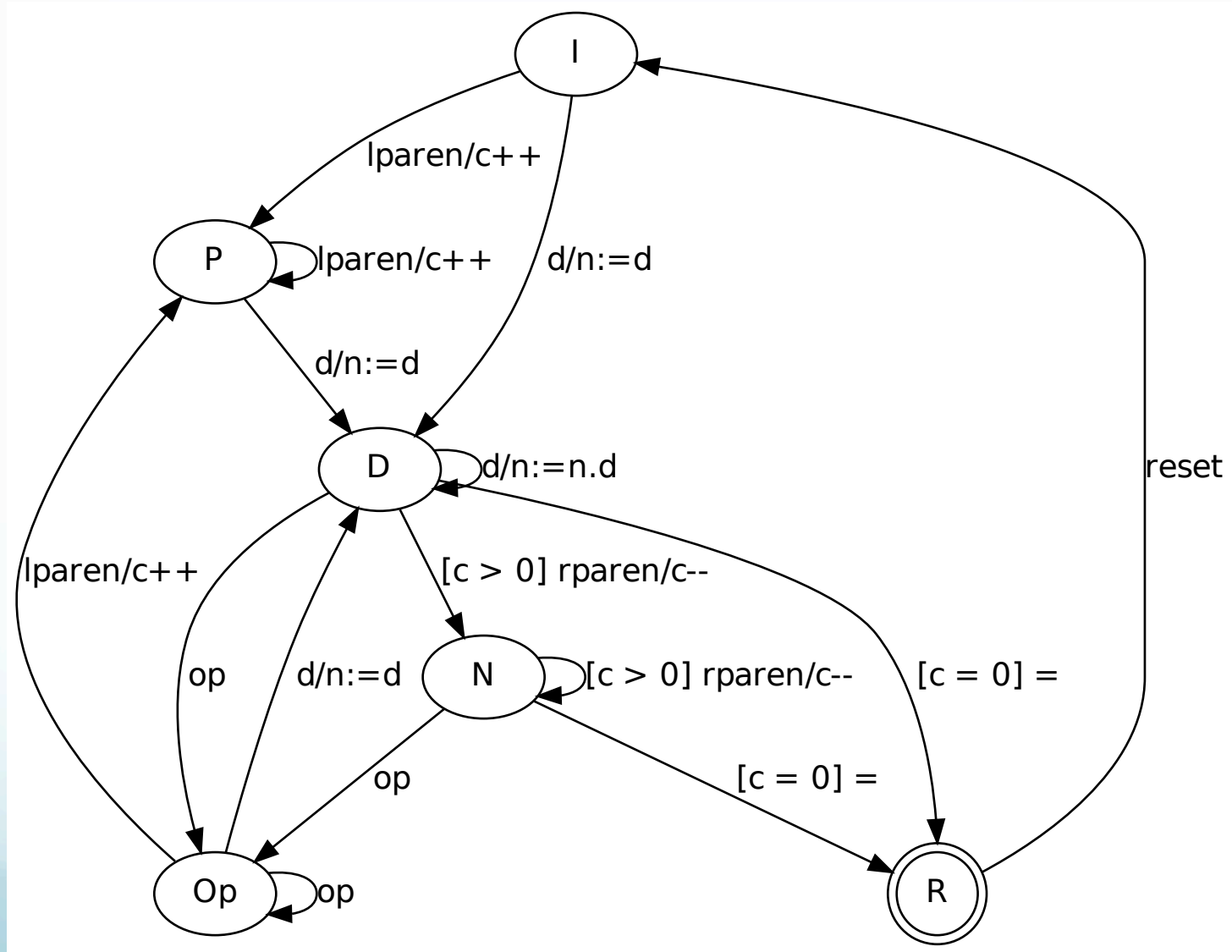


# EFSM for Calculator (v.3)

- Modify the previous calculator by replacing “full integers” by entering digit by digit
- The EFSM should handle digits individually to “build” the integer



# EFSM for Calculator (v.3)



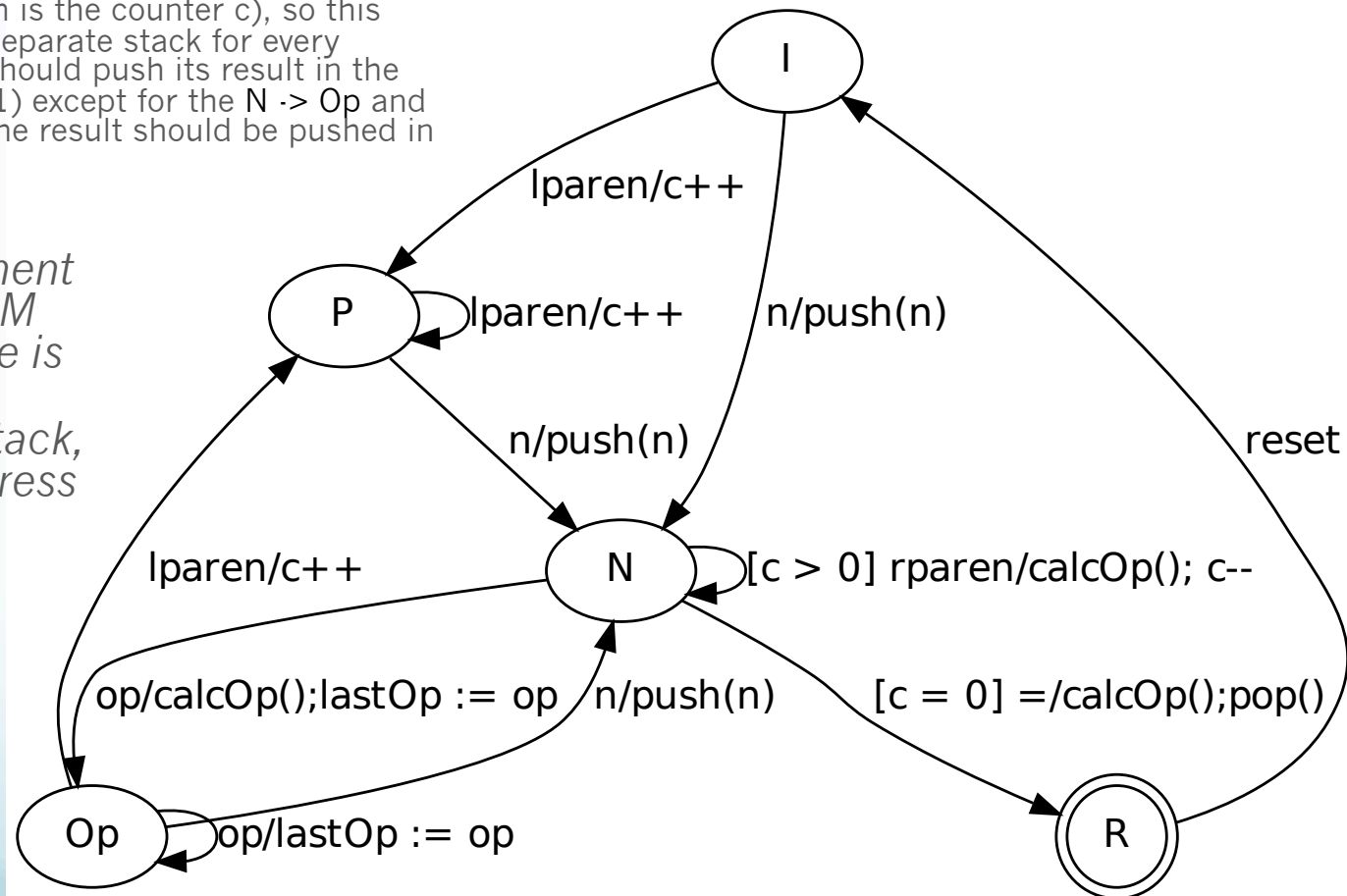
# EFSM for Calculator (v.4)

- Write a more concrete EFSM expressing more operational properties so the evaluation of expressions are done more explicitly
- You should be able to check for division by zero
- Hint: You might use a stack to store operands and to store partial results

# EFSM for Calculator (v.4) - Sketch

- Operands are pushed into a stack as they are read
- The 'current' operator is stored in a variable `lastOp`
- The operation `calcOp` pops two elements off the stack and performs the operation in `lastOp`
- Both `push` and `calcOp` need to be sensitive to the current nesting level (which is the counter `c`), so this implies we should keep a separate stack for every nesting level, and `calcOp` should push its result in the stack of the outer level (`c-1`) except for the `N -> Op` and `N -> R` transitions, where the result should be pushed in the current stack

**Remark:** In the assignment you need a different EFSM (The solution shown here is very much linked to an implementation using stack, and doesn't explicit address division by zero)



# Making your tests executable

- Usually tests extracted from an (E)FSM are quite **abstract** -> need to make them executable
  - The API of the model doesn't match the API of the SUT
- Some common **abstractions** make **difficult** such match
  - Model one aspect of SUT, not whole behavior
  - Omit inputs and outputs which are not relevant
  - Simplify complex data structures
  - Assume SUT is in the correct state for the test
  - Define one model action as representing a sequence of SUT actions
- We must initialize the SUT, add missing details and fix mismatches between the APIs

This **concretization** phase may take as much time as modeling!

# How to Concretize Abstract Tests

- To check SUT outputs we must either:
  - Transform the expected outputs from the model into concrete values
  - Get concrete outputs from the SUT and transform them into abstract values at the model

Some issues:

- Objects in SUT -> must keep track of identity (not only values)
- Need to maintain a map between abstract and concrete objects
  - Each time model creates a new abstract value  $A$  -> SUT creates a concrete object  $C$  (add pair  $(A,C)$  to the map table)
- Different approaches to do so...

# How to Concretize Abstract Tests

- **Adaptation**: Write a wrapper (**adapter**) around the SUT to provide a more abstract view of SUT
- **Transformation**: Transform abstract tests into concrete test scripts

# The Adaptation Approach

- The **adapter** code act as an **interpreter** for **abstract operation calls of model**, executing them in SUT (on-the-fly while abstract tests are generated)

**Adapters** responsible for:

- **Setup**: configuring and initializing the SUT
- **Concretization**: translate model abstract operation call (and inputs) into SUT concrete calls (and inputs)
- **Abstraction**: translate back concrete results into abstract values to the model
- **Teardown**: shut down SUT at end of each test suite, to prepare for next test suite

# The Transformation Approach

- **Test scripts** are produced in the **transformation** approach to transform each abstract test into an executable one

What is needed:

- Setup and teardown code at the beginning and end of each test sequence
- A complex template: many SUT operations to implement 1 abstract operation; trap SUT exceptions to check whether expected or not, etc
- A mapping from each abstract value to a concrete one
- A complex test script with conditionals to check SUT outputs when non-determinism



# Which Approach is Better?

- **Adaptation** better for **online testing**
  - Tightly integrated, two-way connection between MBT tool and SUT
- **Transformation** has the advantage of producing tests scripts in the same language (same naming, structure) as used in manual tests
  - Good for offline testing (less disruption)
- For **offline testing** good to combine both (*mixed*)
  - Abstract tests transformed into executable test scripts which call an adapter layer to handle low-level SUT operations

# Online Testing in ModelJUnit

## Example: Set<String>

Implementation of Set<String> (see assignment 4)

- **StringSet.java**
  - A simple implementation of a set of strings
- **SimpleSet.java**
  - A simplified model of a set of elements
  - Only the model (no adapter): could be used to generate offline tests
  - The model assumes a set with maximum two elements
- **SimpleSetWithAdaptor.java**
  - Like SimpleSet but with adapter code
  - Allow to do online testing of a Set<String> implementation

**Note:** In the following slides we do not include the "import" packages - See the distribution for full code

# Online Testing in ModelJUnit

## Implementation: StringSet

```
public class StringSet extends AbstractSet<String>
{ private ArrayList<String> contents = new ArrayList<String>();
```

```
    @Override
    public Iterator<String> iterator()
    { return contents.iterator(); }
```

```
    @Override
    public int size()
    { return contents.size(); }
```

```
    @Override
    public boolean equals(Object arg0)
    { boolean same = false;
      if (arg0 instanceof Set) {
        Set<String> other = (Set<String>) arg0;
        same = size() == other.size();
        for (int i = contents.size() - 1; same && i >= 0; i--) {
          if (!other.contains(contents.get(i)))
            same = false; } }
      return same; }
```

```
    @Override
    public void clear()
    { contents.clear(); }
```

```
    @Override
    public boolean contains(Object arg0)
    { for (int i = contents.size() - 1; i >= 0; i--) {
      if (contents.get(i).equals(arg0))
        return true; } // return immediately
      return false; } // none match
```

```
    @Override
    public boolean isEmpty()
    { return contents.size() == 0; }
```

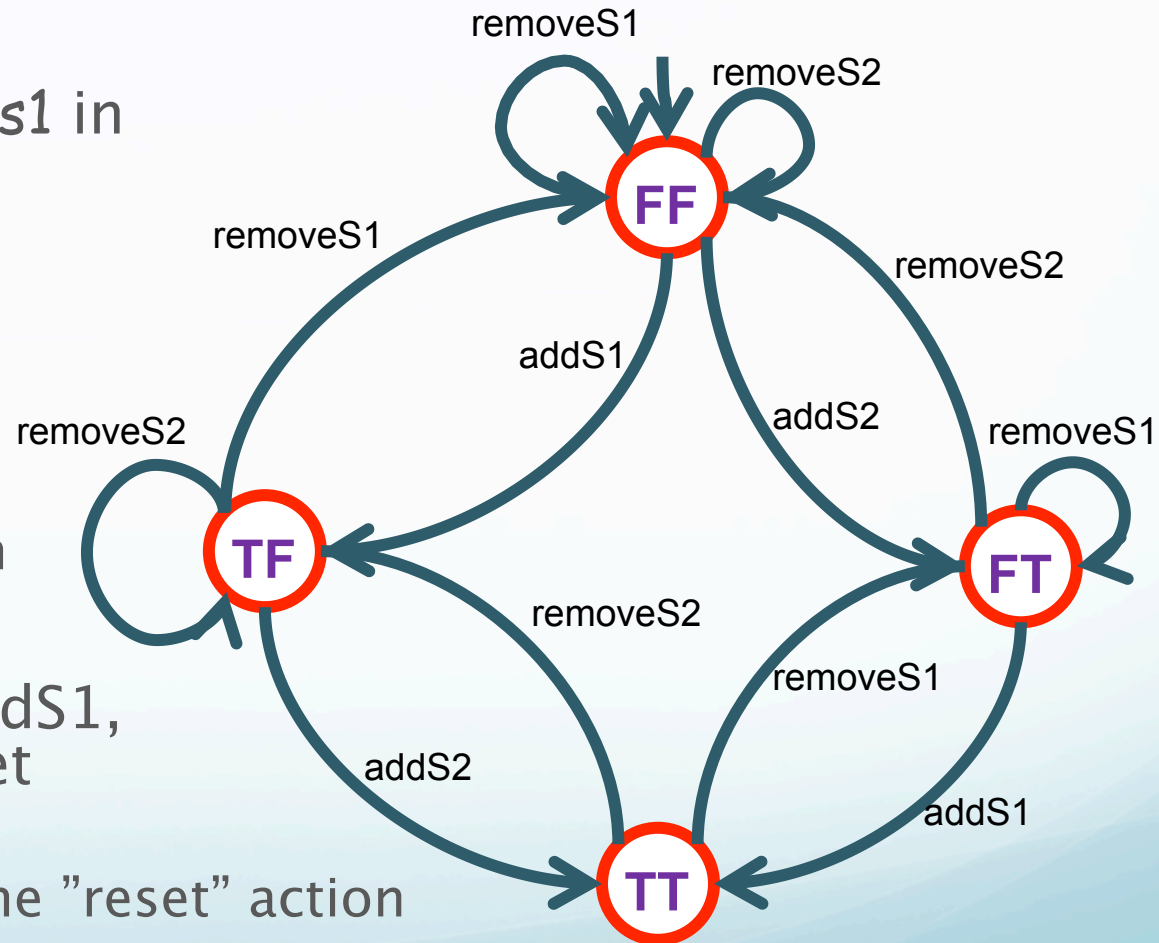
```
    @Override
    public boolean add(String e)
    { if (e == null) {
      throw new NullPointerException(); }
      if (contents.contains(e)) {
        return false; }
      else {
        return contents.add(e); } } // always adds to end
```

```
    @Override
    public boolean remove(Object o)
    { if (contents.isEmpty())
      return false;
      else
        return contents.remove(o); }
  }
```

# Online Testing in ModelJUnit

## EFSM (2-elem set)

- Set:  $S = \{s1, s2\}$
- Representation:  
 $S = \langle x, y \rangle$ , where  $x=T$  if  $s1$  in  $S$  and  $y=T$  if  $s2$  in  $S$
- 4 states:
  - FF  $\rightarrow$   $S$  is empty
  - FT  $\rightarrow$   $S$  contains  $s2$
  - TF  $\rightarrow$   $S$  contains  $s1$
  - TT  $\rightarrow$   $S$  contains both  $s1$  and  $s2$
- Actions: removeS1, addS1, removeS2, addS2, reset



**Note:** we have not added the "reset" action from each state to state FF  
Also, loops with "addS1" ("addS2") are missing

# Online Testing in ModelJUnit

## EFSM: SimpleSet

- So, in the **ModelJUnit** implementation of the set, instead of changing state explicitly, actions simply states how the "internal" variables change
  - **addS1()** -> is applicable only from a state where s1 becomes true
  - **removeS1()** -> is only enable from a state where after applying the action s1 becomes false

# Online Testing in ModelJUnit

## EFSM: SimpleSet

```
public class SimpleSet implements FsmModel  
{ protected boolean s1, s2;
```

```
public Object getState()  
{ return (s1 ? "T" : "F") + (s2 ? "T" : "F"); }
```

```
public void reset(boolean testing)  
{ s1 = false; s2 = false; }
```

```
@Action public void addS1() {s1 = true;}
```

```
@Action public void addS2() {s2 = true;}
```

```
@Action public void removeS1() {s1 = false;}
```

```
@Action public void removeS2() {s2 = false;}
```

```
public static void main(String[] args)  
{ Tester tester = new GreedyTester(new SimpleSet());  
  tester.addListener(new VerboseListener());  
  tester.generate(100);  
}
```

**4 states: TT,  
TF, FT, FF**

**reset transition  
from all states  
to FF**

**Define action to add  
elem S1 to set:  
from any state to  
the state TX**

**Define action to  
remove elem S1:  
from any state to  
the state FX**

**Example to  
generate  
tests from  
the model**

# Online Testing in ModelJUnit

## EFSM with Adaptor: SimpleSetWithAdaptor

```
public class SimpleSetWithAdaptor implements FsmModel
{
    protected Set<String> sut_;
    protected boolean s1, s2;
```

Test data for the SUT

```
    protected String str1 = "some string";
    protected String str2 = ""; // empty string
```

Tests a StringSet implementation (sut\_)

```
public SimpleSetWithAdaptor()
{ sut_ = new StringSet(); }
```

```
public Object getState()
{ return (s1 ? "T" : "F") + (s2 ? "T" : "F"); }
```

Concrete operation in SUT for the abstract (EFSM) operation "reset"

```
public void reset(boolean testing)
{ s1 = false;
  s2 = false;
  sut_.clear(); }
```

Concrete operation in SUT for the abstract (EFSM) operation "addS1"

```
@Action public void addS1()
{ s1 = true;
  sut_.add(str1);
  checkSUT(); }
```

Check SUT in right state



# Online Testing in ModelJUnit

## EFSM with Adaptor: SimpleSetWithAdaptor

```
@Action public void addS2()
{ Assert.assertEquals(!s2, sut_.add(str2)); //sut_.add(str2);
s2 = true;
checkSUT(); }

@Action public void removeS1()
{ s1 = false;
sut_.remove(str1);
checkSUT(); }

@Action public void removeS2()
{ Assert.assertEquals(s2, sut_.remove(str2)); //sut_.remove(str2);
s2 = false;
checkSUT(); }

protected void checkSUT()
{ int size = (s1 ? 1 : 0) + (s2 ? 1 : 0);
Assert.assertEquals(size, sut_.size());
Assert.assertEquals(s1, sut_.contains(str1));
Assert.assertEquals(s2, sut_.contains(str2));
Assert.assertEquals(!s1 && !s2, sut_.isEmpty());
Assert.assertEquals(!s1 && s2, sut_.equals(Collections.singleton(str2))); }

public static void main(String[] args)
{ Set<String> sut = new StringSetBuggy(); // StringSetBuggy();
Tester tester = new GreedyTester(new SimpleSetWithAdaptor(sut));
tester.addListener(new VerboseListener());
tester.addCoverageMetric(new TransitionCoverage());
tester.generate(50);
tester.printCoverage(); } }
```

How to test the result of sut\_.add(...)

Concrete operation in SUT for the abstract (EFSM) operation "removeS1"

Check SUT in expected state

Check size of model and implementaion is the same

If EFSM in state where s2=T, then implement. should be in state where str2 is in the set

Example of generating tests from this model



# Online Testing in ModelJUnit

## Additional Remarks

- ModelJUnit, an iterative process:

getstate() ->  
evaluate guard ->  
execute action ->  
update internal state ->...

- At each moment it is possible to relate with the SUT and check its state through the adapter

- You can add code to measure coverage, traverse the model using specific algorithms, etc
- The code is automatically added when using the "Test Configuration" in ModelJUnit
- In some applications you have to modify the code too (not in the StringSet example)

# Assignment 6

You will have to:

- Define the EFSM of a complex calculator
- Encode it in ModelJUnit
- Write an adapter
- Execute online tests to find errors, using some of ModelJUnit traversal algorithms
  - Define (and measure) state and transition coverage

# Next lecture...

- No lecture on Wednesday (only exercises in the afternoon as usual)
- Next lecture on Monday May 13

# References

- M. Utting and B. Legeard, *Practical Model-Based Testing*. Elsevier - Morgan Kaufmann Publishers, 2007
  - Sections 5.3 and 8.1