Early Experiences on Model Transformation Testing

Alessandro Tiso - Gianna Reggio – Maurizio Leotta
DIBRIS
Università di Genova, Italy

Contents

- Considered problems
- Case study
- Model transformation testing: two approaches
  - Target execution analysis
  - Checking target static properties
- Test suite
    - How to build models used for testing model transformation
- Regression Tests
- Conclusion and future work

Context

- The Model Transformation object of our testing was developed as an application of a general **Me**thod for **D**eveloping **M**odel **T**ransformations (*MeDMT)*
- It was developed during the last year of a three-year PhD course as an application of MeDMT
- We wanted to test this model transformation

Model Transformation Testing issues

- Model Transformation Language heterogenity
  - We chose:
    - Two model transformation languages
- Building a good set of input models for testing purposes
  - We define a criteria to build input models for this purpose

Model Transformation Testing issues

- Definition of oracle functions is difficult
  - We analyse semantic and sinctactic properties of the transformation target
- Support tools and their integration
  - We chose a technology environment

Case Study

- A Model Transformation that:
  - Input
    - UML Design Models built following MARS method
  - Output
    - complete java desktop application (excluding the GUI) in the form of a Java project managed by Maven
- AutoMARS is our tool that implement the model transformation of the case study

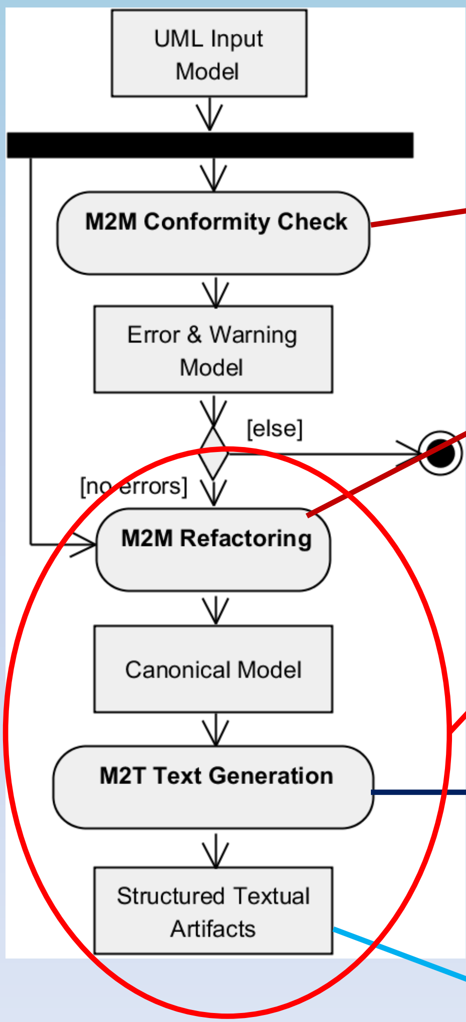
Case Study
input Model

- Profiled UML Models
  - <<context>>, active classes that represent the entities external to the application interacting with it;
  - <<boundary>>, active classes that represent entities taking care of the interaction of the system with some context entities;
  - <<executor>>, active classes that represent entities performing some core system activities;
  - <<store>>, passive classes that represent entities containing persistent data
  - Other stereotypes…
- Only a subset of UML with well defined semantics

Case Study
output

- Complete Java desktop application (exluding the GUI) in the form of a Java project managed by Maven
- Application is built using:
  - Spring as glue framework
  - JPA with Hibernate as persistence provider
  - OCL expressions are compiled in Java

Case Study
Model Transformation Architecture



One Model-to-Model Transformation written in ATL (eclipse)

Four Model-To-Model Refinement Transformations written in ATL (eclipse)

Transformation under test

One Model-to-Text Transformation written in Eclipse Acceleo

Maven project containing code and configuration

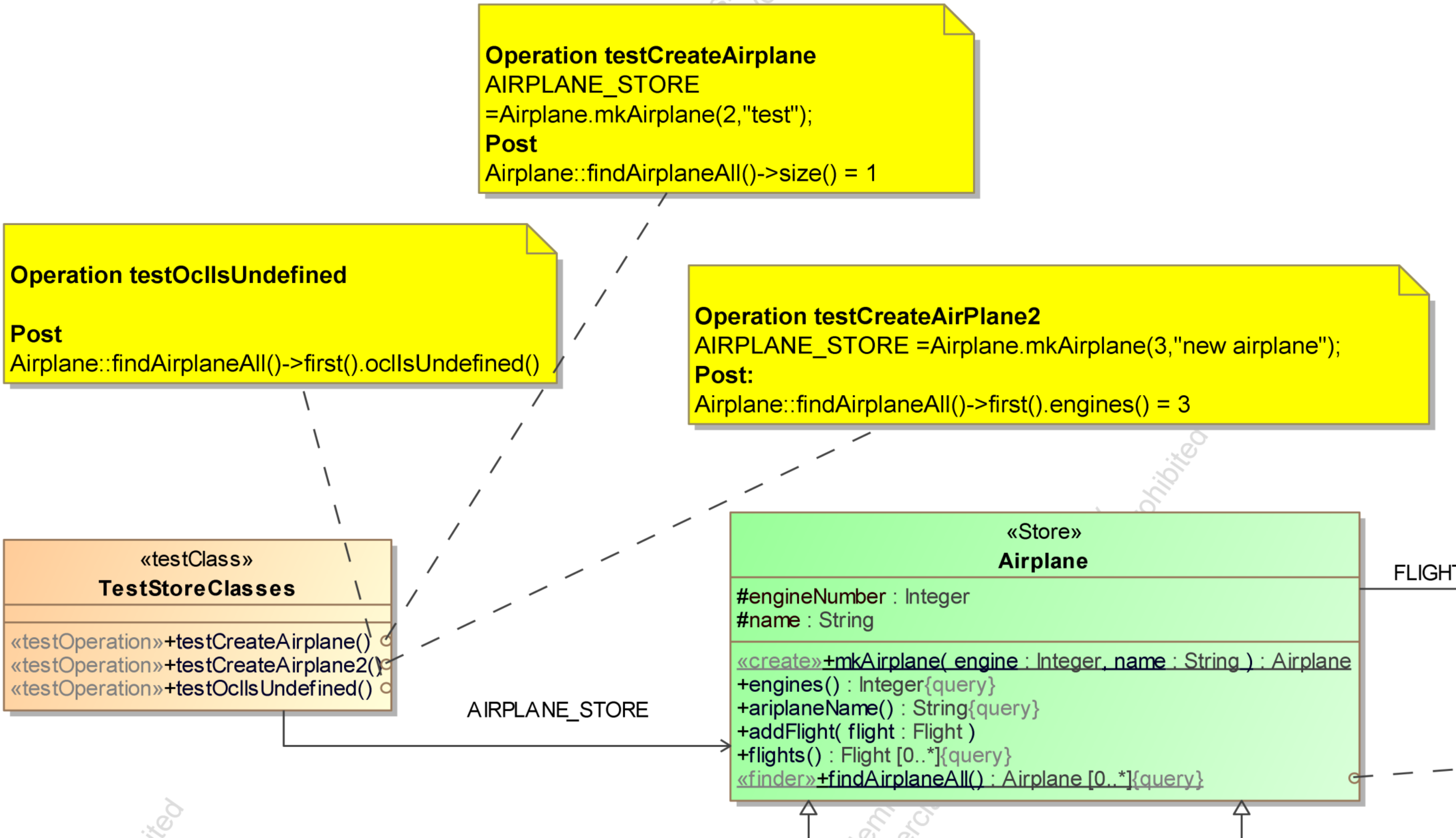
First Approach
Transformation Target execution analysis

- Only if transformation target is executable
  1. Compiling target
  2. Insert in the source model test classes test and operations
     - Generate executable test cases in the target
     - Execute the test cases in the target

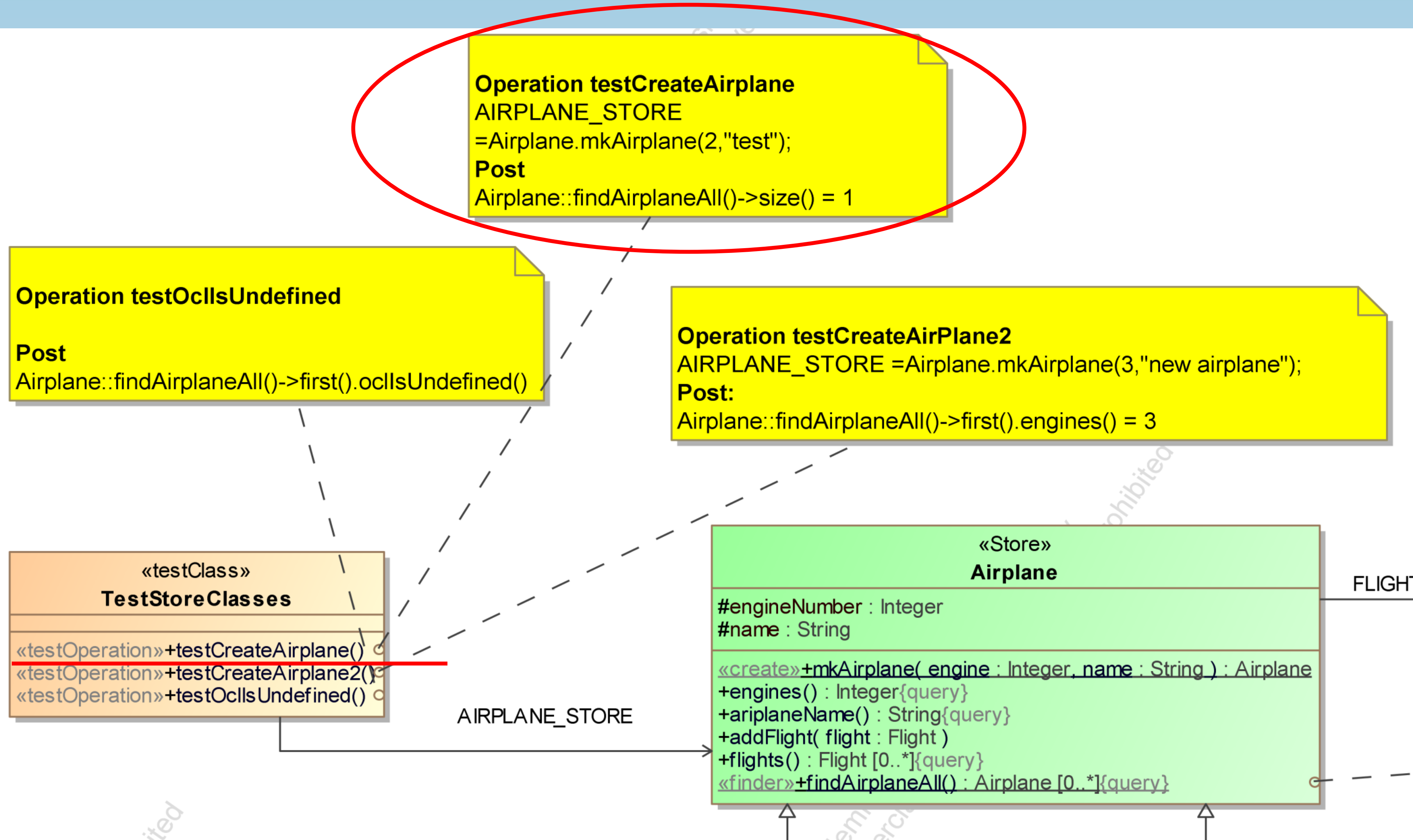
First Approach
Transformation Target execution analysis

- When test are executed
  - Execution of test in the target fail
    - Bug in the model?
    - Bug in the transformation?
    - Bug in the model AND Bug in the transformation
      - More investigation is needed
  - Execution of test in the target succeeds
    - The transformation is bug free
      - Excluding the case of two errors compensating each other
- We need very simple behaviour of tests
  - Qualitatively speaking
    - p(model bug) << p(transformation bug)

Target execution analysis
Store example



Target execution analysis
Store example



Store Example Implementation

Operation testCreateAirplane
AIRPLANE_STORE
=Airplane.mkAirplane(2,"test");
Post
Airplane::findAirplaneAll()->size() = 1

```
@Test
@Transactional
@Rollback(true)
public void testTestCreateAirplane(){
…
classRef = new TestStoreClasses();
      try{
        classRef.testCreateAirplane();
      }catch(Exception e){
exceptionOccurred = true;
…
}
```

```
public class TestStoreClasses  {
    …
private void testCreateAirplaneBody(){
  AIRPLANE_STORE=Airplane.mkAirplane(2,"test");
}
…
private Boolean testCreateAirplanePostCondition(){
     Boolean cond=true;
     if(!( Airplane.findAirplaneAll()
            .size().equals(Integer.valueOf(1)))){
      cond = false;
      }
   return cond;
}
…
 public  void testCreateAirplane() {
   testCreateAirplaneBody();
    if(!testCreateAirplanePostCondition()) {
      throw new PostConditionException("operation:
testCreateAirplane ");
     }
}
```

Compiled OCL

Store Example Implementation

```
Operation testCreateAirplane
AIRPLANE_STORE
=Airplane.mkAirplane(2,"test");
Post
Airplane::findAirplaneAll()->size() = 1
```

```
@Test
@Transactional
@Rollback(true)
public void testTestCreateAirplane(){
…
classRef = new TestStoreClasses();
     try{
        classRef.testCreateAirplane();
     }catch(Exception e){
exceptionOccurred = true;
…
}
```

```
public class TestStoreClasses  {
    …
private void testCreateAirplaneBody(){
  AIRPLANE_STORE=Airplane.mkAirplane(2,"test");
}
…
private Boolean testCreateAirplanePostCondition(){
     Boolean cond=true;
     if(!( Airplane.findAirplaneAll()
           .size().equals(Integer.valueOf(1)))){
      cond = false;
      }
   return cond;
}
…
 public  void testCreateAirplane() {
   testCreateAirplaneBody();
    if(!testCreateAirplanePostCondition()) {
      throw new PostConditionException("operation:
testCreateAirplane ");
     }
}
```

Compiled OCL

Store Example Implementation

Operation testCreateAirplane
AIRPLANE_STORE
=Airplane.mkAirplane(2,"test");
Post
Airplane::findAirplaneAll()->size() = 1

```
@Test
@Transactional
@Rollback(true)
public void testTestCreateAirplane(){
...
classRef = new TestStoreClasses();
      try{
         classRef.testCreateAirplane();
      }catch(Exception e){
exceptionOccurred = true;
...
}
```

```
public class TestStoreClasses  {
    ...
private void testCreateAirplaneBody(){
  AIRPLANE_STORE=Airplane.mkAirplane(2,"test");
}
...
private Boolean testCreateAirplanePostCondition(){
     Boolean cond=true;
     if(!( Airplane.findAirplaneAll()
            .size().equals(Integer.valueOf(1)))){
      cond = false;
      }
   return cond;
}
...
 public  void testCreateAirplane() {
   testCreateAirplaneBody();
    if(!testCreateAirplanePostCondition()) {
      throw new PostConditionException("operation: 
testCreateAirplane “);
     }
}
```

Compiled OCL

Store Example Implementation

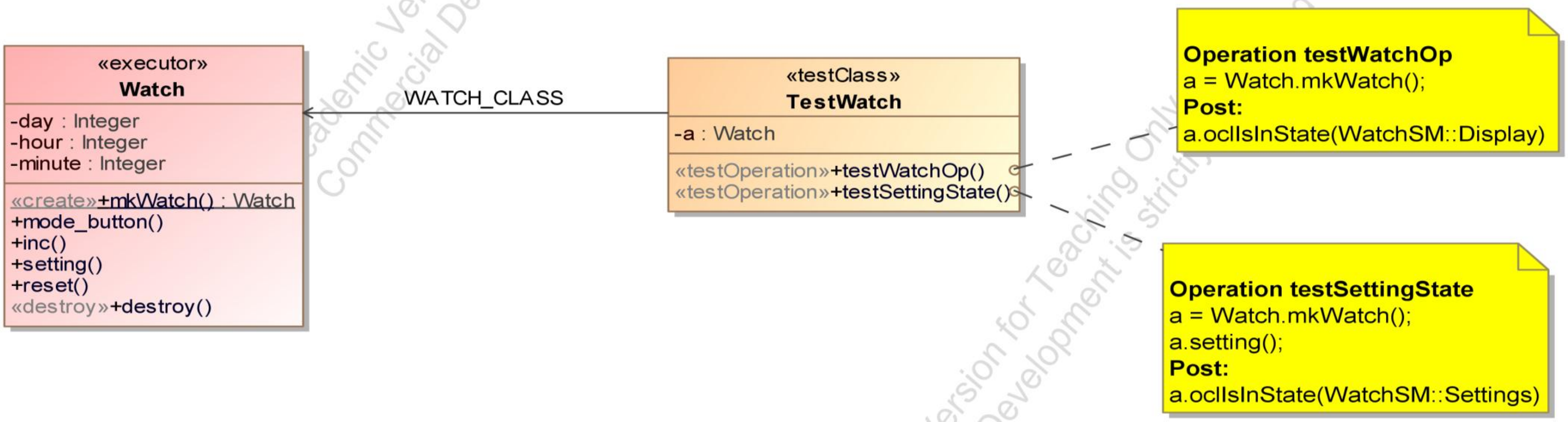
Operation testCreateAirplane
AIRPLANE_STORE
=Airplane.mkAirplane(2,"test");
Post
Airplane::findAirplaneAll()->size() = 1

```
@Test
@Transactional
@Rollback(true)
public void testTestCreateAirplane(){
...
classRef = new TestStoreClasses();
     try{
        classRef.testCreateAirplane();
     }catch(Exception e){
exceptionOccurred = true;
...
}
```

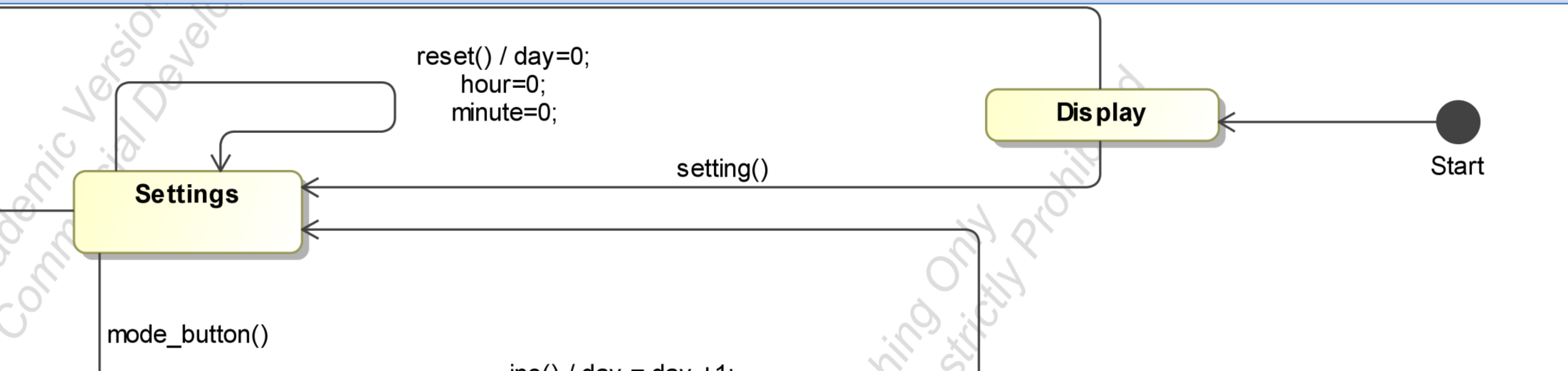
```
public class TestStoreClasses  {
   ...
private void testCreateAirplaneBody(){
  AIRPLANE_STORE=Airplane.mkAirplane(2,"test");
}
...
private Boolean testCreateAirplanePostCondition(){
     Boolean cond=true;
     if(!( Airplane.findAirplaneAll()
           .size().equals(Integer.valueOf(1)))){
      cond = false;
      }
   return cond;
}
...
 public  void testCreateAirplane() {
  testCreateAirplaneBody();
   if(!testCreateAirplanePostCondition()) {
     throw new PostConditionException("operation:
testCreateAirplane ");
    }
}
```

Compiled OCL

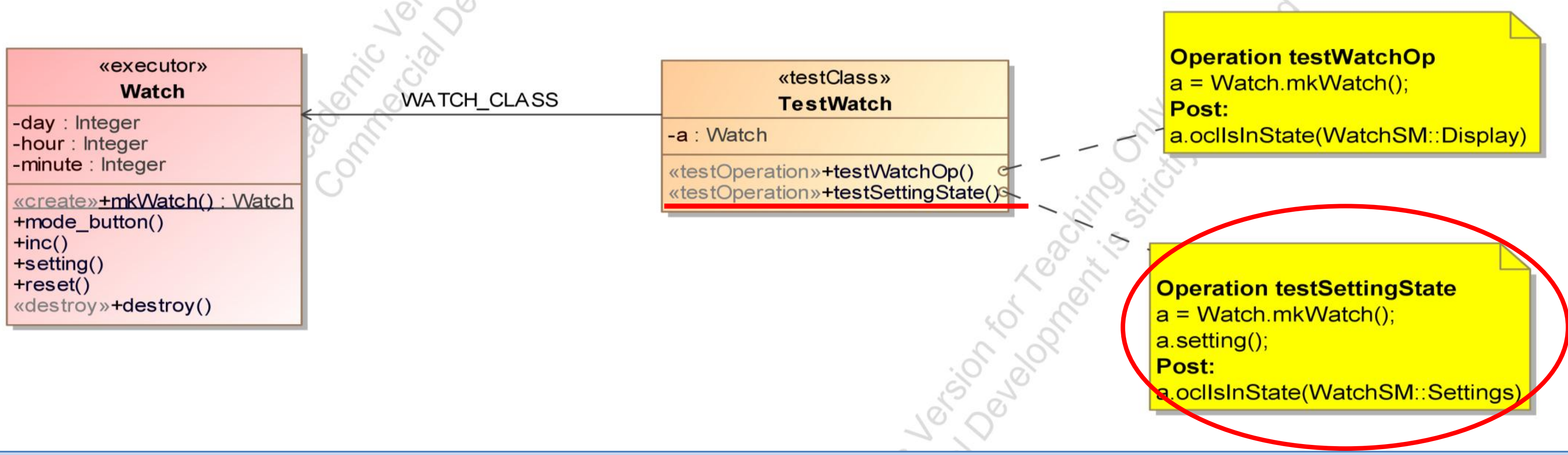
Target execution analysis executor example



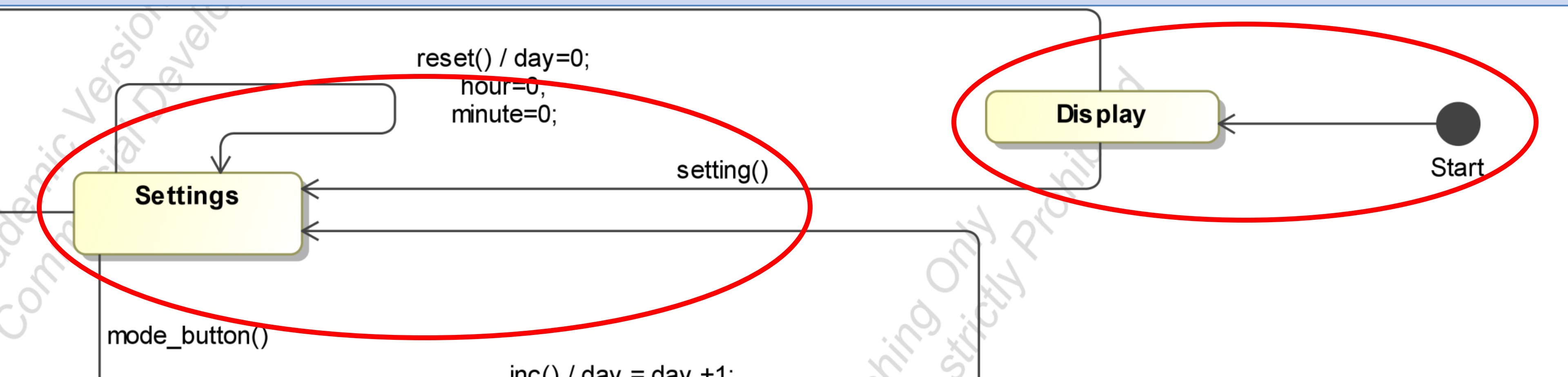
Fragment of the state machine defining the behaviour of Watch class



Target execution analysis executor example



Fragment of the state machine defining the behaviour of Watch class



Executor Example Implementation

Operation testSettingState
a = Watch.mkWatch();
a.setting();
Post:
a.oclIsInState(WatchSM::Settings)

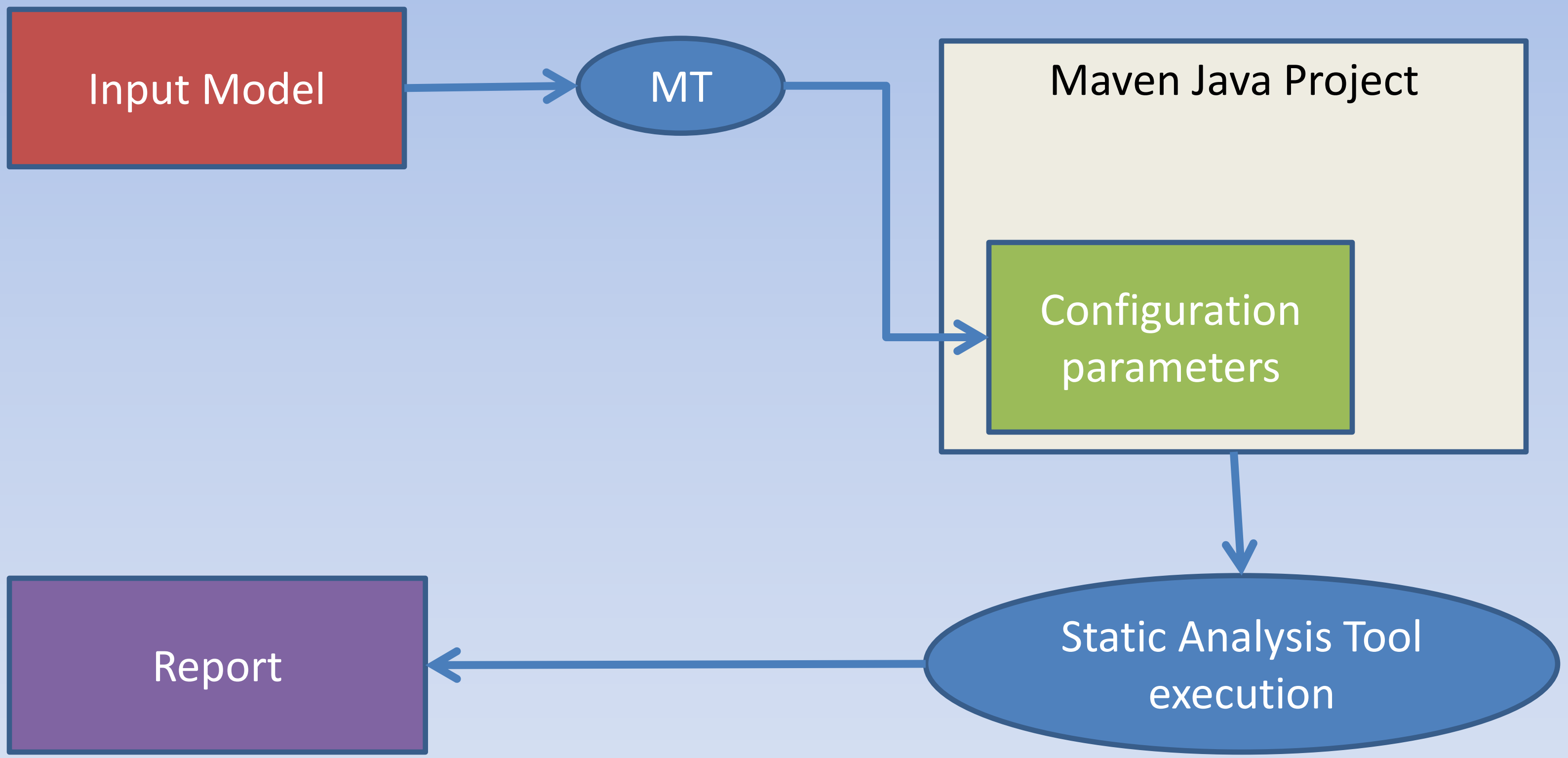
```
@Test
 public void testTestSettingState(){
  ...
   classRef = new TestWatch();
   try{
      classRef.testSettingState();
   }catch(Exception e){
     exceptionOccurred = true;
     ...
   }
}
```

```
public class TestWatch extends AbstractActiveClassTest
{
  private void testSettingStateBody(){
    a = Watch.mkWatch();
    a.setting();
    }
...
  private Boolean testSettingStatePostCondition(){
   Boolean cond=true;
    if(!( this. a.oclIsInState ("WatchSM::Settings", this ))){
     cond = false;
    }
    return cond;
  }
...
public  void testSettingState() {
  testSettingStateBody();
  if(!testSettingStatePostCondition()) {
   throw new PostConditionException("operation: testSettingState ");
  }
}
```

Second approach
Checking static properties

- Assess the presence of specific elements in the target
- Analysing the input model
  - Compute text snippets that must be in the transformation target
- Analysing the target
  - Assert the presence of text snippets in the target

Checking static properties in the case study



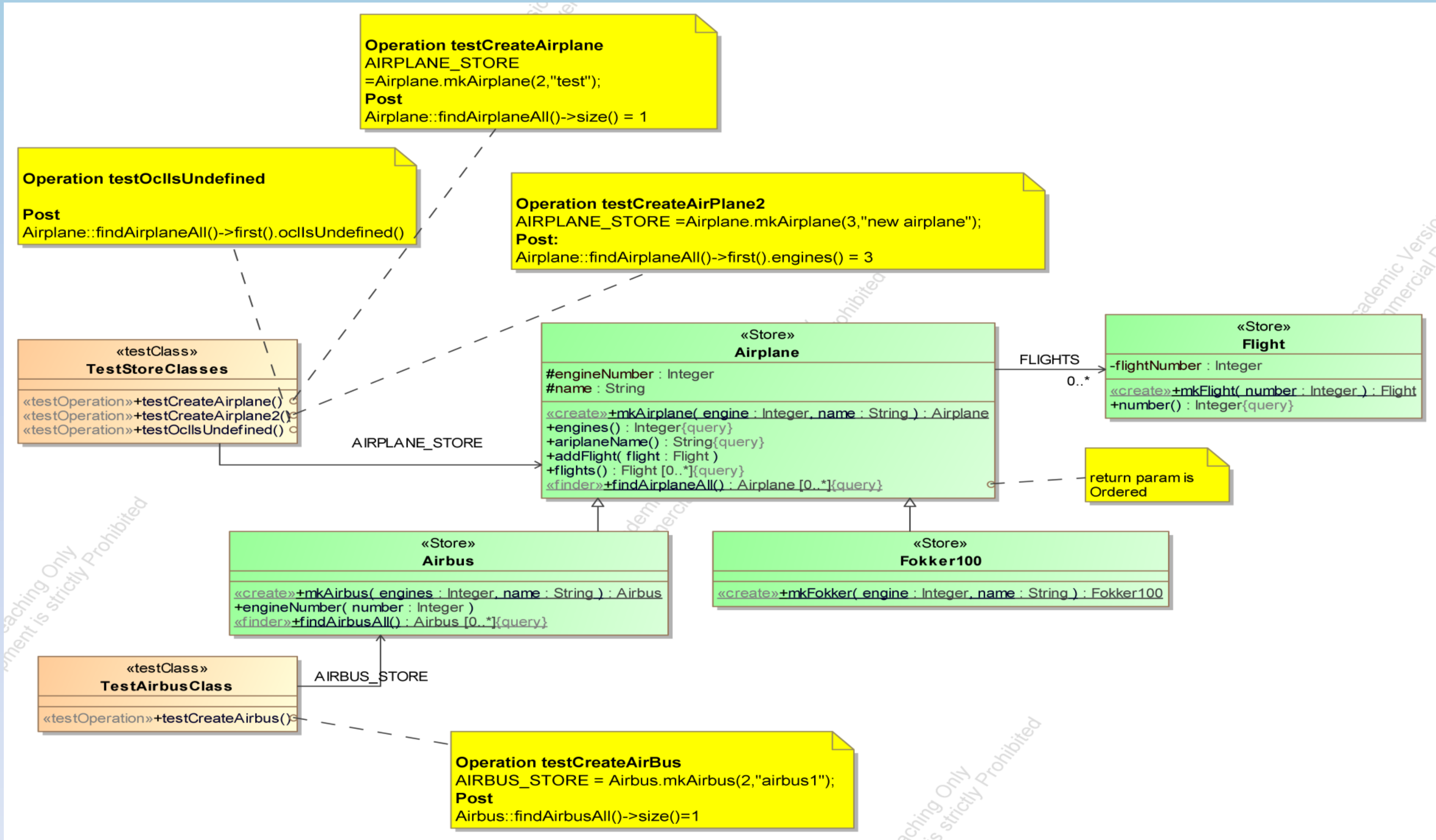
Test Suite
How to build test models

- Test models manually written
- Small input models containing mainly only one kind of input elements
  - One input model for each stereotype
  - Each pattern used in the clauses defining the model transformation design should be instantiated

Input Models used for testing

- Four test models
  - Data Type
    - containing mainly data types
  - Executor
    - containing mainly executors
  - Boundary
    - containing mainly boundaries
  - Store
    - containing mainly stores
- Each one containing:
  - Test Classes and test operations
- We have also a model containing all the stereotypes used in the other models

Test Models used
Store

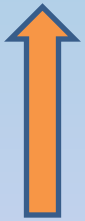


Regression Test

- Compares the output of a specific run of the model transformation with the expected one
  - White spaces and line breaks are not considered
- Each model of the test suite activate only a subset of the modules composing the model transformation
- Is useful only when
  - New features are added
  - The model transformation is refactorized

Conclusion

- Usefulness
  1. Analysing the target execution
  2. Regression test
  3. Checking static properties



HIGH

LOW

- Using hand made small input models containing mainly one kind of stereotypes has simplified the bugs finding activity
- Simple tools and techniques are very important developing "real" model transformations

Future Work

- Generalize MeDMT giving guidelines for building:
  - Input test models
  - Test cases on the result of the trasformation starting from the design of the transformation itself
- Execute some experiments to asses the effectiveness of our approaches

Thank you for your attention