SOME EXPERIMENTS ON TEST CASE TRACEBAILITY

Macario Polo, Beatriz Pérez and Pedro Reales
Department of Information Systems and Technologies
University of Castilla-La Mancha
Paseo de la Universidad, 4
13071-Ciudad Real (Spain)
{macario.polo,beatriz.plamancha,pedro.reales}@uclm.es

Abstract.
This paper investigates the relationships between the test cases generated from state machines and their ability to find faults in the implementation of the class proceeding from the state machine. Many research works have proposed strategies to generate test cases for different kinds of software artefacts. However, to our best knowledge, the "traceability" of the coverage got by the test cases in different abstraction levels has not been studied, so being this work the first (or one of the first) contribution on this sense.

1. INTRODUCTION

Testing is a process which involves all the activities of the software life cycle. Besides the V-Model, which proposes the verification and validation of all the artefacts developed during the construction of a software system, different authors have proposed techniques for preparing test cases from the early stages of software development [1-10], some of which are reviewed in the next section of this paper.

Basanieri et al. [1], for example, derive test cases for use cases from the sequence diagrams corresponding to the scenarios of the use case, translating the sequence of messages to test templates, which may be later combined with actual test data to get executable test cases. Thus, from the diagram shown in Figure 1 (taken from the referenced work), the test templates obtained by the authors start with messages with no predecessor (i.e., launched by actors) whose execution is continued by the dependent messages (such as: *start.open(); enterUserName(String); enterPassword(String); loginUser().validateUser(String, String).setUpSecurityContext()*). Guards may imply the introduction of alternative messages: note in the figure that, after setting up the security context, the possible success of the operation may determine the execution either of *newUserId* or *closeLoginSelection.* When the test engineer has the test templates, he/she combines them with actual data in order to get test cases which likely will be able to be executed against the system implementation.

Other authors have described techniques and coverage criteria to get test cases from other types of diagrams. Thus, Andrews et al. [10] propose a set of coverage criteria for UML class and interaction diagrams, which help to know whether a test suite is adequate with respect to a given coverage criterion: for example, if an association has *p..n* as cardinality on one of its ends, the test suite should instantiate exactly *p, n, p+1* and *n-1* elements. In the same way, the Andrews et al.'s coverage criteria for interaction diagrams could be used to check the quality of the test cases produced with the Basanieri et al.'s method

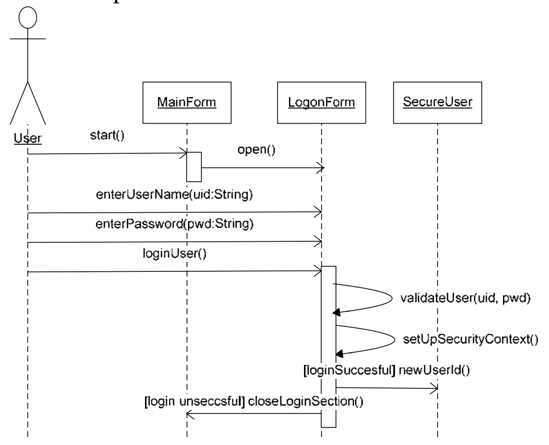
User
MainForm
LogonForm
SecureUser
start()
open()
enterUserName(uid:String)
enterPassword(pwd:String)
loginUser()
validateUser(uid, pwd)
setUpSecurityContext()
[loginSuccesful] newUserId()
[login unseccsful] closeLoginSection()

Figure 1. A sample sequence diagram (taken from [1])

.These types of techniques emphasize the importance of testing from the beginning of software projects, so being close to the idea of continuous verification and validation, a broad concept which includes "testing". But, in the specific case of the two techniques so briefly summarized here, they also help to prepare the source code of test cases which, probably, will be executable when a version of the system is available.

It is evident, therefore, the immersion in the "testing culture" brought by this kind of approaches, what is a significant benefit for software practitioners and academics. However, it is not clear the relationship between the quality of a test suite defined for a UML artefact (such as a sequence, class or state diagram) and the quality reached by that very same suite on the executable code corresponding to that original artefact.

This paper presents a first contribution on this sense, with the analysis of the quality that test cases generated from state machines get on source code. The idea is represented in Figure 2: given the behaviour of a class, specified by means of a state machine, and a state-machine criterion-adequate test suite got by the application of some technique, the goal is to study whether the test suite is still adequate when an executable specification of the class is available: this is, does remain the coverage traceable when the abstraction level of a software artefact is decreased?

Since this work is focused on statecharts, a revision of some works on state machines and test case generation from state machines is presented in Section 2. Then, Section 3 describes two case studies and discusses the results obtained. Finally, Section 4 draws our conclusions and future lines of work.

2. Related work

In the current UML specification [11], state machines have two main uses: (1) behavioural state machines, and (2) protocol state machines. Since we are interested in studying the correspondence between models and code, for this context we will focus on the first use, which is also illustrated in Figure 2 with the *Has a behaviour defined in* relationship (which corresponds to the relationship between the Class and StateMachine UML classifiers [12]).

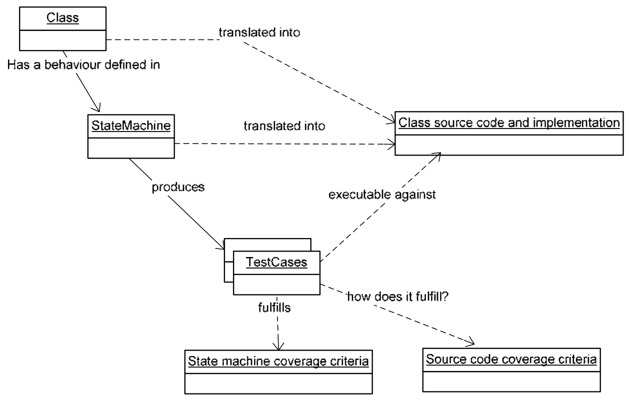
Class
translated into
Has a behaviour defined in
StateMachine
translated into
Class source code and implementation
produces
executable against
TestCases
fulfills
how does it fulfill?
State machine coverage criteria
Source code coverage criteria

Figure 2. Schema of this proposal

The figure also shows the *translation* relationship between the specification of a class (whose behaviour is annotated with a state machine) and its implementation. Related to this translation, Warmer and Kleppe [13] informally describe an algorithm to translate class and state machines to source code. They say, for example:

> When the transitions are implemented by operations, i.e., the event at the transition is actually an operation call to which the object responds, the guard should be considered part of the precondition of the operation. In that case, the start state should also be considered part of the precondition. The end state is considered part of the postcondition. Implementing them follows the rules for implementing pre- and postconditions.

According to their rules, the source state can be considered part of the precondition of the implementation of the transitions triggered by operation *calls*, whilst the target state could be part of the postcondition. If the transition has also a guard condition, this is considered part of the precondition. Furthermore, if the same event occurs for more than one transition, all possible situations must be taken into account in order to correctly write the preconditions, because it must be checked that the class instance is in an accepted state for triggering that transition.

Since actually a class state can be described as a function of the class fields' values, it is easy to get a source code specification from a state machine. If the mechanism to know the instance state is costly, Warmer and Kleppe propose to add a field representing the state to the class, which must be recalculated when any of the influencing fields changes, and which is queried before triggering the transitions. Figure 3 shows the partial specification of a supposed banking Account with some states and transitions. Some of these ones are UML *Call events*, since they correspond to operations (*deposit* and *withdraw*) of the Account class.

An account may be in the state *Zero*, *ZeroOrPositive* or *Negative* depending on the value of one or more of its fields (i.e., *balance)*. As it is seen, *withdraw* can be executed when the instance is in the ZeroOrPositive or in Negative. Thus, a simple code of *withdraw* could be written such as in Figure 4. As Warmer and Kleppe suggest, this code could include other additional sentences to check that the instance reaches the expected state after executing the operation, so representing postconditions.

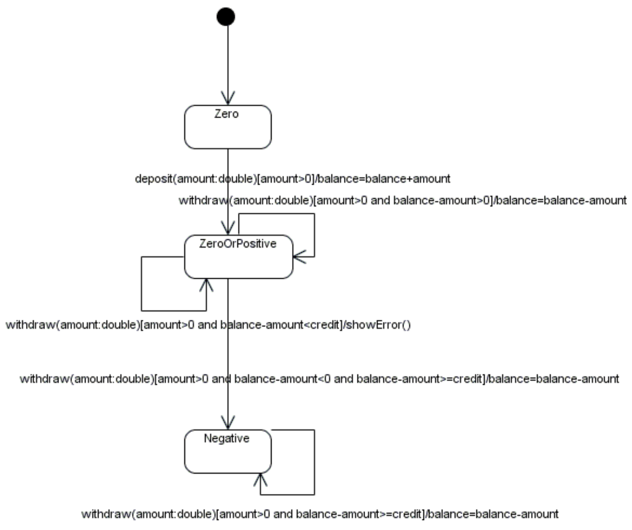
Zero
deposit(amount:double)[amount>0]/balance=balance+amount
withdraw(amount:double)[amount>0 and balance-amount>0]/balance=balance-amount
ZeroOrPositive
withdraw(amount:double)[amount>0 and balance-amount<credit]/showError()
withdraw(amount:double)[amount>0 and balance-amount<0 and balance-amount>=credit]/balance=balance-amount
Negative
withdraw(amount:double)[amount>0 and balance-amount>=credit]/balance=balance-amount

Figure 3. Partial specification of a banking Account

```
public void withdraw(double amount) {
  if (state==ZeroOrPositive) {
     if (amount>0 && balance-amount<credit)
        showError();
     else if (amount>0 && balance-amount>0)
        balance=balance-amount;
     else if (amount>0&&balance-amount<0&&balance-amount>=credit)
        balance=balance-amount;
  } else if (state==Negative) {
     if (amount>0 && balance-amount>=credit)
        balance=balance-amount;
  }
}
```

Figure 4. Possible code of *withdraw*

Figure 5 shows a simplified algorithm, inspired in [13], to get the code from a class state machine: for each transition, it finds the corresponding method in the class description and builds a new *block of sentences* (for composing the precondition and the possible effect) which is added to the method. In a previous work related with a tool for database reverse engineering and code generation [14], we successfully used this very same algorithm to generate code for classes which proceed from database tables: the classes and their relationships are inferred from tables and foreign key relationships, and the software engineer may modify the default behaviour (provided by the tool) of the classes with the addition of one state machine to each class. The figure also includes the structure of both a Class as a StateMachine, which is also compatible with Figure 2.

```
Class=(Name, Fields, Operations, SateMachine)
operation=(Name, Parameters, Return,
    BlocksOfSentences)

StateMachine=(States, Transitions, Initial∈
    States, End∈ States)
t ∈ Transitions=(Call, Guard, Effect)

function getCodeFromSM(class : Class) {
    ∀ t ∈ class.StateMachine.transitions {
        op=class.findOperation (t.call)
        Block b=new Block()
        b.addPrecondition(t.sourceState)
        b.addPrecondition(t.guardCondition)
        b.addBody(t.effect)
        op.add(b)
    }
}
```

Figure 5. An algorithm for generating code from a state machine

In summary, it is clear that: (1) the behaviour of a class can be formally described by means of a state machine; and (2) there is an effective means to generate the code corresponding to a class annotated with a state machine.

In order to know whether a test suite is adequate for testing a software artefact, the definition of one or more coverage criteria for such type of artefact is required. In this sense, Hong et al. [8] and Offutt et al. [15] have proposed the following coverage criteria for state machines:

- State coverage [8]. A test suite *T* satisfies *state coverage* if each state is covered by one or more test sequences in *T*.
- Transition [8, 15]. A test suite *T* satisfies this criterion if each transition is traversed by one or more test sequences in *T*.
- Full predicate [15]. For each predicate *P* on each transition and each test clause c_i in *P*, *T* must include tests that cause each clause c_i in *P* to determine the value of *P*, where c_i has both the values *true* and *false*. A predicate is a boolean expression whose value may determine the triggering of a transition.
- Transition pair [15]. For each pair of adjacent transitions $S_i : S_j$ and $S_j : S_k$, *T* must contain a test that traverses each transition of the pair in sequence.
- Configuration [8]. According to these authors, a configuration is "maximal set of states in which a system can be simultaneously". Thus, a test suite *T* satisfies this criterion if each configuration is covered by one or more test cases in *T*.
- Complete sequence [15]. *T* must contain tests that traverse meaningful sequences of transitions on the state machine. "Meaningful sequences" are chosen by the test engineer based on experience, domain knowledge and other human-based knowledge.

There exist subsumption relationships among criteria: for example, Transition pair subsumes Transitions, which in turn subsumes the State criterion. A criterion coverage C_1 subsumes another criterion C_2 if for every program, any test set *T* that satisfies C_1 also satisfies C_2 [16].

Once a coverage criterion has been selected, the following step is the generation of test cases by the application of some algorithm which, in general, my be adopted from graph literature. Also, a state machine can be understood as a finite automata [17] and be processed by means of regular expressions, whose alphabet is composed by the set of operations that trigger the transitions. In a previous work [18], we have shown a tool that generates test cases based on regular expressions.

3. Analysis of test cases traceability

With "traceability", we reference the degree that coverage reached in a software artefact described at a given, high abstraction level, is preserved when such artefact is translated into a lower software artefact.

The work is focused on the traceability of test cases from state machines to executable specifications, for which we have used two experiments, one extracted from literature and one which has been elaborated by us.

1) The first is the Cruise Control system, which has been used, among others, in references [15, 19, 20], where the corresponding state machine can be found.
2) The second one corresponds to a Manager that controls the light flow of two semaphores, and has been developed by these authors as a problem to be resolved by their students (Figure 6). When there are no pedestrians, the manager changes the light of both semaphores (*a* and *b*) sending them the *change* event every a fixed number of seconds (60, 63, 66, 83, and 86). However, a pedestrian may request the red light in any of the semaphores: if the semaphore where red is requested is in yellow or red, nothing happens; if it is in green and the semaphore is *a*, then the managers changes *a* to yellow either 20 seconds after the request or, if less than 20 seconds remain, in this time. If the red light is requested on *b*, then the request is passed to *a*.

For both cases we have applied State, Transition and Transition pair coverage. Then:

1) The two state machines have been translated into Java programs, according to the afore-mentioned translation rules [13], and the test cases for the three coverage criteria were generated.

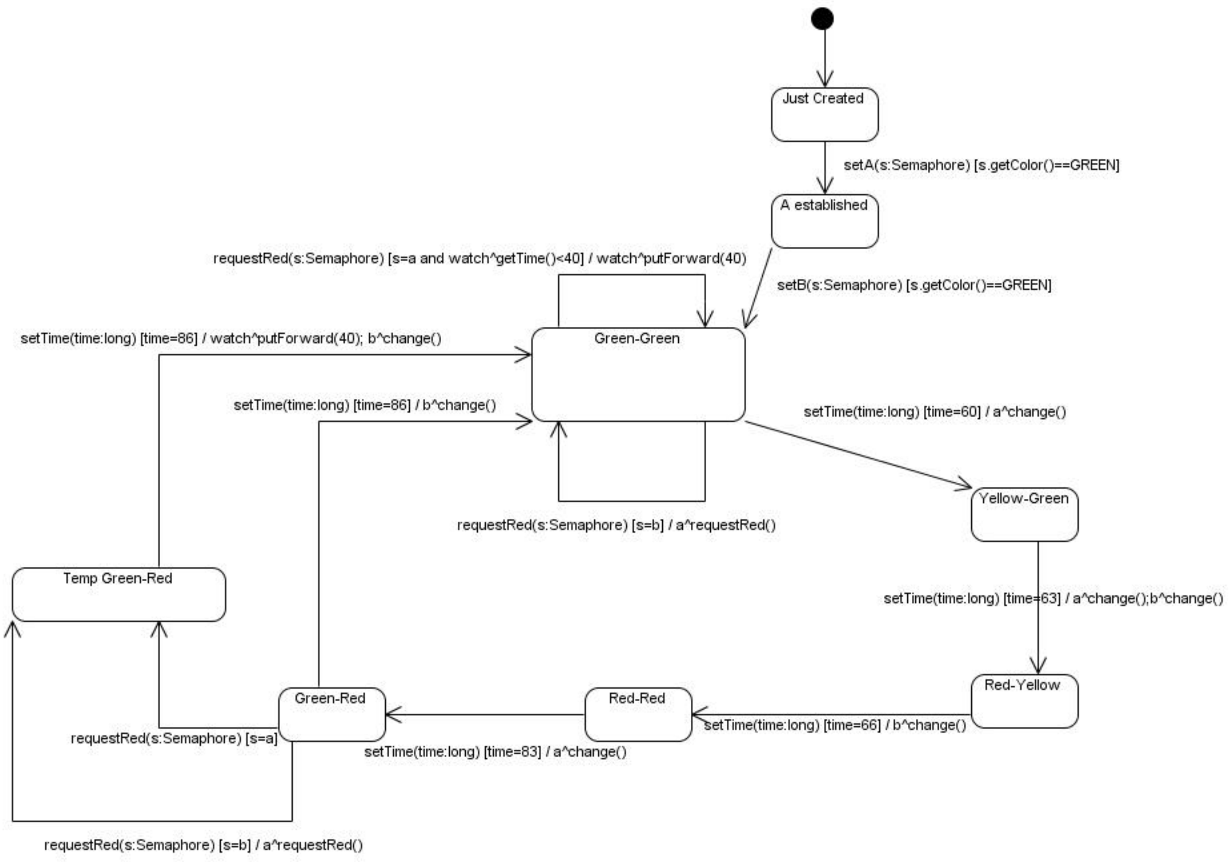
Just Created
setA(s:Semaphore) [s.getColor()==GREEN]
A established
requestRed(s:Semaphore) [s=a and watch^getTime()<40] / watch^putForward(40)
setB(s:Semaphore) [s.getColor()==GREEN]
setTime(time:long) [time=86] / watch^putForward(40); b^change()
Green-Green
setTime(time:long) [time=86] / b^change()
setTime(time:long) [time=60] / a^change()
Yellow-Green
requestRed(s:Semaphore) [s=b] / a^requestRed()
Temp Green-Red
setTime(time:long) [time=63] / a^change();b^change()
Red-Yellow
Green-Red
Red-Red
requestRed(s:Semaphore) [s=a]
setTime(time:long) [time=66] / b^change()
setTime(time:long) [time=83] / a^change()
requestRed(s:Semaphore) [s=b] / a^requestRed()

Figure 6. State machine for the semaphore's manager example

2) The test cases were then translated into executable Java test cases in MuJava and testooj [18, 21] formats (which are two tools for mutation testing).
3) The coverage reached by the Java test cases has been measured in terms of the mutation score using MuJava [21] (for generating mutants) and testooj [18] (for executing the test cases and analyzing the results).

It its important to note that, since each mutant represents a faulty version of the program under test, the assessment of the mutation score is an excellent predictor of the ability of the test suite for finding faults.

3.1. Getting the source code

The state machines have been faithfully translated into Java programs according to the algorithm presented in Figure 5. The code generated is freely available for download [22]. As an example, Figure 7 shows the source code of the *requestRed* method in the semaphores' manager, which is one of the operations accepted by the Manager class (according to its state machine, Figure 6): each transition labelled with *requestRed* in the state machine is collected with a precondition, which proceeds from the corresponding source state and the possible guard condition.

3.2. Getting the test cases

State and transition test cases have been manually written, whilst transition pair cases have been generated with a simple program that goes through the state machines with that kind of route. They are shown in Figure 8. This very same program translates them into JUnit and MuJava test cases.

```
public void requestRed(Semaphore semaphore) {
    if (a.getColor()==GREEN && b.getColor()==GREEN) {
        if (semaphore==a) {
            long currentMiliseconds=this.watch.getCurrentMiliseconds();
            if (currentMiliseconds<40000) {
                this.watch.putForward(40000);
            }
        } else
                this.a.requestRed();
    } else if (a.getColor()==GREEN && b.getColor()==RED) {
        if (semaphore==a) {
            this.temporalState=true;
        } else
            this.a.requestRed();
    }
}
```

Figure 7. Source code of *Manager::requestRed*

3.3. Quality of the executable test cases

To check the ability of the test suitse to find faults, we have generated mutants with MuJava. To minimize the cost of testing, *testooj* was used to reduce the mutant suites by applying second-order mutation [23], a technique which reduces the size of the mutant set to a half, reduces much more the number of functionally equivalent mutants and accelerates the mutant execution and the result analysis steps. Then, the test cases were executed against the second-order mutants using *testooj*.

Table 1 shows the number of second-order mutants generated and the results of the execution of the tests against the mutants, expressed in terms of the mutation score. Note that in all the cases, the results obtained are far from the optimal score, which is 100%.

		Mutation score		
Class	**2nd order mutants**	**State**	**Transition**	**Transition pair**
CruiseControl	56	35%	46%	83%
Manager	47	61%	63%	80%

Table 1. Results obtained in the two experiments

3.4. Analysis of results

State coverage requires that each state in the machine is traversed by a test case. This implies that just one of the transitions arriving the state is executed, maybe existing transitions corresponding to other methods that are not executed. Therefore, in terms of source code coverage, state coverage does not even imply method coverage (which is got when all methods in a class are executed at least once).

Regarding Transition coverage, it means that each transition is triggered at least once, what usually implies the execution of each method of the corresponding class: transitions correspond to public methods, whose *effect* clause may involve the call to a private method. Thus, coverage of transitions implies coverage of methods

The best results of Table 1 correspond to Transition pair coverage. According to the algorithm shown in Figure 5, when a transition pair *(p,q)* is traversed by a test case on the state machine, this means that the preconditions for *p*, *q* (*source states + guard conditions*) have been fulfilled. This leads to that the executable test case forces the execution of the corresponding *true branch* in the conditional instruction (in the semaphores' manager, for example, a test case goes from Green-Green to Red-Yellow, leaving Yellow-Green with *time=63).* The test case corresponding to the *false branch* is not generated if the state machine does not contemplate it (in the same example, the manager is not tested when it is in Yellow-Green and the *setTime* method is triggered with *time≠63*). Thus, there may be test situations which, being almost completely tested on the state machine, are not tested enough on its corresponding implementation. This criterion implies partial All decisions coverage (All decisions is got when each decision in the source code is executed with *true* and *false).*

4. CONCLUSIONS AND FUTURE WORK

This paper has presented an initial study about the keeping of test cases coverage between different representations of the same system, expressed at different abstraction levels. It has focused on thee coverage criteria for class state machines and their corresponding source code, and has been applied to two single case studies.

In order to get better results, more experimentation with other systems is required. Ideally, these experiments should be carried out with benchmark systems (such as [24]), that allow to other researchers the replication of the experiments and the advance in this research line, that we believe quite interesting to a better understanding of the need of applying validation and testing techniques along the whole life cycle.

The investigation of test cases traceability is also required with other kinds of software artefacts, such as class and sequence diagrams, which is a research line we are now also starting.

Pair transition test cases for *CruiseControl* (ign=ignition; activ=activate; brk=brake; dct=dct; res=resume; tF=tooFast)
[new, ign, ign, ign, act, tF, ign -]
[new, ign, ign, ign, act, tF, act, brk, res, tF -]
[new, ign, ign, ign, act, tF, act, brk, res, brk, act, tF -]
[new, ign, ign, ign, act, tF, act, brk, res, brk, act, brk, res, tF -]
[new, ign, ign, ign, act, tF, act, brk, res, brk, act, brk, res, brk, ign, ign -]
[new, ign, ign, ign, act, tF, act, brk, res, brk, act, brk, res, brk, brk, res, deact, res, ign, ign -]
[new, ign, ign, ign, act, tF, act, brk, res, brk, act, brk, res, brk, brk, res, deact, act, deact, res, deact, ign -]
[new, ign, ign, ign, act, tF, act, brk, res, brk, act, brk, res, brk, brk, res, deact, act, deact, res, deact, brk, act, ign -]
[new, ign, ign, ign, act, tF, act, brk, res, brk, act, brk, res, brk, brk, res, deact, act, deact, res, deact, brk, res, ign -]
[new, ign, ign, ign, act, tF, act, brk, res, brk, act, brk, res, brk, brk, res, deact, act, deact, res, deact, brk, ign -]
[new, ign, ign, ign, act, tF, act, brk, res, brk, act, brk, res, brk, brk, res, deact, act, deact, res, deact, brk, brk -]
[new, ign, ign, ign, act, tF, act, deact -]
[new, ign, ign, ign, act, tF, act, ign -]

Pair transition test cases for the semaphores' *Manager* (sT=setTime; rR=requestRed; watch.gT=watch.getTime; GR=GREEN)
[new, setA(a) [s.getColor()==GR], setB(b) [s.getColor()==GR], sT(60), sT(63), sT(66), sT(83), sT(86), sT(60)]
[new, setA(a) [s.getColor()==GR], setB(b) [s.getColor()==GR], sT(60), sT(63), sT(66), sT(83), sT(86), rR(a) [watch.gT()<40000], sT(60)]
[new, setA(a) [s.getColor()==GR], setB(b) [s.getColor()==GR], sT(60), sT(63), sT(66), sT(83), sT(86), rR(a) [watch.gT()<40000], rR(a) [watch.gT()<40000], rR(b), sT(60)]
[new, setA(a) [s.getColor()==GR], setB(b) [s.getColor()==GR], sT(60), sT(63), sT(66), sT(83), sT(86), rR(a) [watch.gT()<40000], rR(a) [watch.gT()<40000], rR(b), rR(a) [watch.gT()<40000]]
[new, setA(a) [s.getColor()==GR], setB(b) [s.getColor()==GR], sT(60), sT(63), sT(66), sT(83), sT(86), rR(a) [watch.gT()<40000], rR(a) [watch.gT()<40000], rR(b), rR(b)]
[new, setA(a) [s.getColor()==GR], setB(b) [s.getColor()==GR], sT(60), sT(63), sT(66), sT(83), sT(86), rR(b)]
[new, setA(a) [s.getColor()==GR], setB(b) [s.getColor()==GR], sT(60), sT(63), sT(66), sT(83), rR(a), sT(86), sT(60)]
[new, setA(a) [s.getColor()==GR], setB(b) [s.getColor()==GR], sT(60), sT(63), sT(66), sT(83), rR(a), sT(86), rR(a) [watch.gT()<40000]]
[new, setA(a) [s.getColor()==GR], setB(b) [s.getColor()==GR], sT(60), sT(63), sT(66), sT(83), rR(a), sT(86), rR(b)]
[new, setA(a) [s.getColor()==GR], setB(b) [s.getColor()==GR], sT(60), sT(63), sT(66), sT(83), rR(b), sT(86)]
[new, setA(a) [s.getColor()==GR], setB(b) [s.getColor()==GR], rR(a) [watch.gT()<40000]]
[new, setA(a) [s.getColor()==GR], setB(b) [s.getColor()==GR], rR(b)]

Figure 8. Transition pair test cases

5. ACKNOWLEDGMENTS

This work is partially supported by the PRALÍN (Pruebas en Líneas de Producto) project, financed by the Government of Castilla-La Mancha and the European Social Fund, by the grant number PAC08-0121-1374.

6. REFERENCES

1. Basanieri F, Bertolino A and Marchetti E (2002). *The Cow_Suite Approach to Planning and Deriving Test Suites*

in UML Projects. 5th International Conference on The Unified Modeling Language: Springer-Verlag. LNCS.
2. Ball T, Hoffman D, Ruskey F, Webber R and White L (2000). *State generation and automated class testing.* Software Testing, Verification and Reliability, 10, p. 149-170.
3. Baudry B, Traon YL and Sunyé G (2002). *Testability Analysis of a UML Class Diagram*. 8th IEEE Symposium on Software Metrics.
4. Burton S, Clark J and McDermid J (2001). *Automatic generation of tests from statechart specifications*. Formal Approaches to Testing of Software (FATES'01). Aalborg, Denmark: BRICS: Basic Research in Computer Science.
5. Chow T (1978). *Testing software designs modeled by finite-state machines.* IEEE Transactions on Software Engineering, 4(3), p. 178-187.
6. Edwards SH (2000). *Black-box testing using flow-graphs: an experimental assessment of effectiveness and automation potential.* Software Testing, Verification and Reliability, 10(4), p. 249-262.
7. Grieskamp W, Gurevich Y, Schulte W and Veanes M (2001). *Testing with abstract state machines*. Formal Methods and Tools for Computer Science. Canary Islands, Spain.
8. Hong HS, Lee I and Sokolsky O (2001). *Automatic test generation from statecharts using model checking*. Formal Approaches to Testing of Software (FATES'01). Aalborg, Denmark: BRICS: Basic Research in Computer Science.
9. Tse T and Xu Z (1996). *Test Case Generation for Class-Level Object-Oriented Testing*. 9th International Software Quality Week. San Francisco, CA.
10. Andrews A, France R and Ghosh S (2003). *Test adequacy criteria for UML design models.* Software Testing, Verification and Reliability, (13), p. 95-127.
11. OMG (2005). Unified Modeling Language: Superstructure version 2.0. Object Management Group.
12. OMG (2005). UML 2.0 OCL Specification. Object Management Group.
13. Warmer J and Kleppe A (2003). *The Object Constraint Language, 2nd edition*: Addison Wesley.
14. Polo M, García-Rodríguez I and Piattini M (2007). *An MDA-based approach for database reengineering.* Journal of Software Maintenance & Evolution: Research and Practice, 19(6), p. 383-417.
15. Offutt AJ, Liu S, Abdurazik A and Amman P (2003). *Generating test data from state-based specifications.* Software Testing, Verification and Reliability, (13), p. 25-53.
16. Frankl PG and Weyuker EJ (1998). *An applicable familiy of data flow testing criteria.* IEEE Transactions on Software Engineering, 14(10), p. 1483-1498.
17. Kirani S and Tsai WT (1994). *Method sequence specification and verification of classes.* Journal of Object-Oriented Programming, 7(6), p. 28-38.
18. Polo M, Piattini M and Tendero S (2007). *Integrating techniques and tools for testing automation.* Software Testing, Verification and Reliability, 17(1), p. 3-39.
19. Atlee J and Gannon J (1993). *State-based model checking of event-driven system requirements.* IEEE Transactions on Software Engineering, 19(1), p. 24-40.
20. Offutt A (1999). Generating test data from requirements/specifications: Phase II final report. Technical Report ISE-TR-99-01, Department of Information and Software Engineering, George Mason University. Fairfax, VA.
21. Ma Y-S, Offutt J and Kwon YR (2005). *MuJava: an automated class mutation system.* Software Testing, Verification and Reliability, 15(2), p. 97-133.
22. Polo M, Reales P and Pérez B (2009). Laboratory package for Some experiments on test case traceability. Available at (2009, February, 28): http://www.inf-cr.uclm.es/www/mpolo/seke09
23. Polo M, Piattini M and García-Rodríguez I (2008). *Decreasing the cost of mutation testing with second-order mutants.* Software Testing, Verification and Reliability, In press (DOI 10.1002/stvr.392).
24. Do H, Elbaum SG and Rothermel G (2005). *Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact.* Empirical Software Engineering: An International Journal, 10(4), p. 405-435.