Using Model Differencing for Architecture-level Regression Testing

Henry Muccini
Dipartimento di Informatica
University of L'Aquila
Via Vetoio, 1 - L'Aquila, Italy
muccini@di.univaq.it

Abstract

Regression testing can be systematically applied at the software architecture level in order to reduce the cost of retesting modified systems, and also to assess the regression testability of the evolved system.

With the advent of model-based specification and analysis of software architectures, regression testing at the architectural level can be handled by analyzing how the architectural model evolves when moving from an initial to a subsequent version.

This paper analyzes how model differencing, a recent research topic in the model-based community, can be employed for implementing model-based architecture-level regression testing.

1 Introduction

Model-based differencing is being recently studied as a way to deal with evolving models and their subsumed systems: given two models (typically, representing the evolution of the same software system) model differencing algorithms permit to discover similarities and differences among the two models. UMLDiff [25], based on the UML metamodel, considers a graph representation of a model. Given models A and B, the two graphs are traversed and differences are identified according to "name" similarity and "structure" similarity. The SiDiff generic difference algorithm [12], instead, calculates differences of two models given as XML files. UML documents are treated as ordered trees (so to improve performance) and written according to a proposed generic meta-model. Tool specific UML differencing algorithms have been proposed by UML tool vendors, like IBM/Rational [1]. Model differencing techniques, based on model transformation composition, have been proposed in [6].

While model differencing techniques have been proposed so far with the primary intent of reasoning about evolving versions of a given OO system, *goal of this paper is to make use of model differencing techniques for software architecture-based* ***regression testing*** *of Component-based Systems (CBS).*

Goal of an architecture-based testing technique is to increase the confidence on the quality of an assembly of components, by verifying that a system derived by assembling black-box components conforms to architectural decisions [14]. Suites of abstract test cases are selected from the architectural models; test cases are then executed on the CBS so to prove (or disprove) the CBS conformance to its architecture (hereafter referred as CBSA) [5, 20, 7, 19, 9, 21, 11, 14, 15]. Figure 1 summarizes state of the art in architecture-based testing.

Goal of an architecture-based regression testing technique is to reiterate the testing process in a cost effective manner, whenever a CBSA change over time [15]. Test cases are selected by appropriately looking into existing test cases. New test cases can be added.

Goal of this paper is to propose an *architecture-based conformance regression testing approach based on model differences*. The approach is architecture-based since abstract test cases are selected (and re-selected) starting from an architecture-level specification, refined into concrete test cases, and run over the CB implementation. The approach is based on model differences since the algorithm to re-select tests (to be re-executed on the modified architectures) is based on model differences information. While a SA-based regression testing technique shall cope with two main types of evolution (architectural evolution and code evolution, see [15]) this paper focusses on architectural evolution.

The rest of this paper is structured as follows: Section 2 provides some background information on regression testing. Section 3 briefly describes and motivates the approach. Section 4 discusses the developed approach and applies it over a running example. Section 5 briefly describes existing tool support. Section 7 concludes the paper and outlines future work.

33rd EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA 2007)
0-7695-2977-1/07 $25.00 © 2007 IEEE

		Bertolino Inverardi '96 [5]	Richardson Wolf '96 [20]	Eickelmann Richardson '96 [7]	Richardson Stafford Wolf '97 [19]	Harrold '98 [9]	Rosenblum '98 [21]	Jin Offutt '01 [11]	Bertolino Inverardi Muccini et. al. '97-'04 [14]	Muccini, Dias, Richardson '05-'06 [15]
Introduction to the topic		++	++	+	++	+	+	+		
Testing Activities	**Test case Selection**		+		+	++		+	++	++
	Coverage Criteria		++		++	+		++	+	
	Adequacy criteria						++		++	
	Test Execution								++	++
	Results Evaluation				+					+
Testing Phases	**Unit**	++								
	Integration	++		++					++	+
	System									
Testing Goal	**SA-based Testing**		+		+	+		+	++	
	SA assessment		+		+					
Other	**Testability**		+	++	++	++				
	SA styles			+	++					
	Regression Testing	+				++	+			++
	Traceability									

Figure 1. Software Architecture-based Testing Techniques

2 Background: Regression Testing

Regression testing, as quoted from [10], "attempts to validate modified software and ensure that no new errors are introduced into previously tested code". The traditional approach is decomposed into two key phases: i) testing the program P with respect to a specified test suite T, and ii) when a new version P′ is released, regression testing of the modified version P′ to provide confidence that P′ is correct with respect to a test set T′.

To explain how a regression testing technique works in general, let us assume that a program P has been tested with respect to a test set T. When a new version P′ is released, regression testing techniques provide a certain confidence that P′ is correct with respect to a test set T′. In the simplest regression testing technique, called *retest all*, T′ contains all the test cases in T, and P′ is run on T′. In *selective regression testing*, T′ is selected as a "relevant" subset of T, where $t \in$ T is relevant for P′ if there is the potential that it could produce different results on P′ that it did on P (following a *safe* definition).

In general terms and assuming that P is a program under test, T a test suite for P, P′ a modified version of P and T′ the new test suite for P′, regression testing techniques work according to the following steps: 1) select T′, subset of T and relevant for P′; 2) test P′ with respect to T′; 3) if necessary, create T″, to test new functionality/structure in P′; 4) test P′ with respect to T″; 5) create T‴, a new test suite and test history.

All of these steps are important for the success of a selective regression testing technique and each of them involves important problems [8]. However, step 1 (also called, regression test selection) characterizes a selective regression testing technique. For this reason, this paper focusses on this step.

3 Approach Overview and Motivations

Any component-based system has its own architecture (CBSA), in terms of architectural elements and constraints. The CBSA model specifies, by focussing on components integration while hiding implementation details, what to expect (in terms of properties and qualities) from the assembly of components. Since an architectural model may consist of different diagrams, according to the multi-view architectural modeling philosophy [13], we do here focus on structural and behavioral diagrams, that is, those typically utilized for modeling the SA topology (i.e., component diagrams) and its behavior in terms of interacting components and connectors (i.e., sequence and state diagrams).

From the CBSA, architecture-level test cases can be extracted, so to be successively run on the CBS implementation for compliance analysis. When the software architecture evolves, the model documenting the architectural decisions evolves itself. Architectural elements can be added, removed, or modified. The architecture itself can be reconfigured, by changing connectivity among architectural elements. By following the drawing in Figure 2, a new version of the CBSA model is then created.

The approach we are proposing makes use of a *model differencing algorithm* and tool we developed, in order to identify how the structural and behavioral models taken into consideration vary. The output of this model differencing analysis is a delta diagram which is taken in input by a *test re-selection algorithm* in order to identify which test cases need to be re-tested.

This approach inherits from existing model-differencing algorithms the main principles of "name" and "structure"

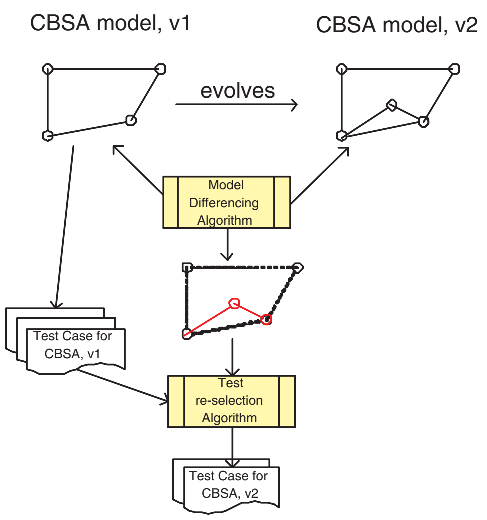


Figure 2. Approach Overview

similarity heuristics, "XML-based difference" calculation, model difference "coloring".

It differs from previous work for two main reasons: we do implement the difference algorithm over state machines (while previous work focus on class diagrams), and it is the first regression testing approach explicitly supported by model differencing algorithms. At the best of our knowledge, while initial reasoning on SA-based regression testing techniques have been proposed in [9], SARTE (Software Architecture-based Regression Testing) is the only framework explicitly dealing with SA-based regression testing [15]. This paper inherits from SARTE the main concepts and extends the test re-selection phase through model differencing. In the domain of model-based regression testing, only very few approaches and tools support regression testing [16] and those tools work at the low level design, not at the architectural level. Indeed, other work on software architecture-based analysis and dependence analysis at the architectural level [23] may be related to this approach.

4 Using Model Differencing for Architecture-level Regression Testing

In the rest of this paper, while assuming the CBSA model is available and architectural test cases have been already selected, we will focus on the model differencing algorithm (used to detect differences and similarities), and to the the test re-selection algorithm (used to identify which architectural test cases are still valid in the modified version of the CBSA).

The rest of the paper will focus on the two algorithms. We initially describe the ATM system to be used as a running example, then propose those two algorithms.

4.1 The ATM running example

We make use of the ATM system as the running example for this paper. While many architecture description languages have been proposed so far for software architecture modeling, in this paper we cast our study on the CHARMY [4, 18] notation for modeling software architectures.

CHARMY allows the specification of a software architecture by means of both a topological (static) description and a behavioral (dynamic) one. To describe the architectural topology, CHARMY uses a subset of the UML component diagram. An architectural component is drawn with the familiar UML 2.0 notation for components. A connector can be seen as a complex coordination element or as a simple communication channel. Complex connectors are modeled using the UML notation for components, while an architectural channel is represented by an association line between architectural components.

The architecture of the ATM system (called ATM_v2) that we consider is composed of four components as shown in Figure 3: the *User*, the transaction manager (*TM* component), the bank account (*BA* component), and the authentication (*AUTH* component). The User component communicates only with the TM component that forwards the service requests to the BA component or to the Auth component.

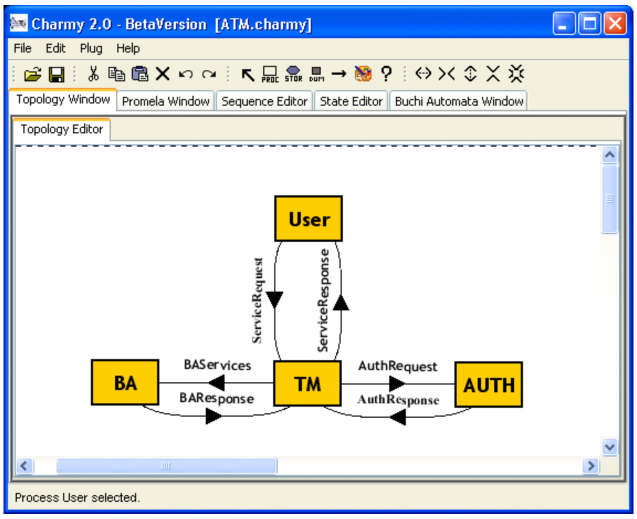
The internal behavior of each component is specified in terms of a State Transition Diagram (STD) notation close to the Promela syntax, defined as a quadruple $(\mathcal{S}, \mathcal{L}, S_0, \mathcal{T})$, where $\mathcal{S}$ is the set of states, $\mathcal{L}$ is the set of distinguished labels (actions) denoting the STD alphabet, $S_0 \in \mathcal{S}$ is the initial state, and $\mathcal{T} = \{\xrightarrow{l} \subseteq \mathcal{S} \times \mathcal{S} \mid l \in \mathcal{L}\}$ is the transition relation labeled with elements of $\mathcal{L}$. The labels $\mathcal{L}$ are structured as follows:

$$[guard]`/`event`(`parameter\_list`)``/`op_1`;`\cdots`;`op_n$$

where *guard* is a boolean condition that denotes the transition activation, an *event e* can be a message sent or received (denoted by an exclamation mark "!*e*" or a question mark "?*e*", respectively) or an internal operation ("*e*"), and can have several parameters as defined in the parameters list. $op_1, op_2, \cdots, op_n$ are the operations to be executed when the transition fires.

The notation is shown in Figure 4: the User component handles three different requests, one for the authentication ($!login$) followed by two possible responses ($?login\_ok$ and $?login\_ko$), one for withdrawing money from her account ($!withdraw$), and one for recharging the mobile phone credit ($!chargePhone$). The TM component contains the logic of the ATM system. This component receives the login request from the User ($?login$) and forwards it to the AUTH component ($!login\_Auth$). Two are the possible responses that TM can receive from AUTH: login success

($?login\_auth\_ok$), and login failure ($?login\_auth\_ko$). In case of success, the User is habilitated to available services (i.e., withdraw money or recharge mobile phone). TM receives the response for both services and forwards them to the User component. The other two components, BA and AUTH, manage the bank account services (i.e., withdraw, charge) and login services, respectively.



Process User selected.

Figure 3. Topology Diagram of the ATM example

From the ATM architectural specification, a suite of seven test specifications have been selected, by applying the approach described in [14]. The selected coverage criteria requires the coverage of each edge in the STD graph. Informally, the test specifications are:

t1: the User tries to login, but her access is denied;

t2: the User logins, then logouts;

t3: the User logins, asks to withdraw money, the connection with her bank account fails and she cannot withdraw money;

t4: the User logins, asks to charge her phone, the connection to the bank account succeed, but she has not enough money in her checking account, and the operation fails;

t5: the User logins, asks to charge her phone, and succeed.

t6: the User logins, asks to withdraw money, the connection with her bank succeed, but he has not enough money for the withdraw;

t7: the User logins, asks to withdraw money, and succeeds.

As remarked before, those are abstract test specifications (as in most of the model-based testing approaches) to be successively (i.e., when the CBS implementation becomes available) refined into concrete and executable test cases. They can be selected and specified according to existing model-based and architecture-based testing techniques [16, 14].

4.2 Model Differencing

While the comparison between topology diagrams is a straightforward instantiation and adaptation of existing algorithms, to keep the section more focussed, we here analyze how we compared the CHARMY state machines.

When dealing with state machines, two are the main elements to be compared: states and transitions. As shown in the excerpt of the CHARMY XML schema for state diagrams in Figure 5, a state $\mathcal{S}$ in the STD graph (*ELEMENTSTATE* tag) is identified by an *ID*, a *NAME*, and a *TYPE* (initial or internal state). A transition label $\mathcal{L}$ has a more informative structure, with an *ID*, a *NAME*, a transition *TYPE* (synchronous, asynchronous, or loop), the transition *SOURCE*, the transition *TARGET*, the *ACTIONS* (events), the *CONDITIONS* (guards), and *PARAMETERS*.

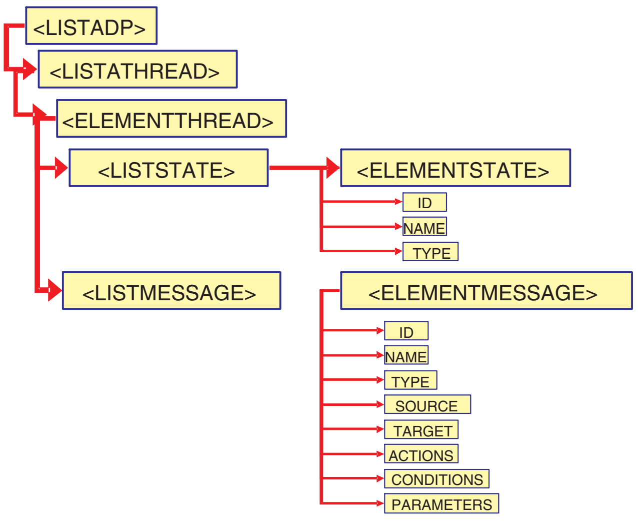


Figure 5. The CHARMY state machines XML schema

When comparing two CHARMY state diagrams, the algorithm takes in input the two STD graphs and outputs an STD_{diff} differencing graph highlighting unchanged and changed elements. The algorithm initially compares *ELEMENTSTATE* tags for state names similarity. STD_1 *ELEMENTSTATE* tags are traversed and compared with STD_2 states. Two states, S_i in STD_1 and S_j in STD_2, are the same

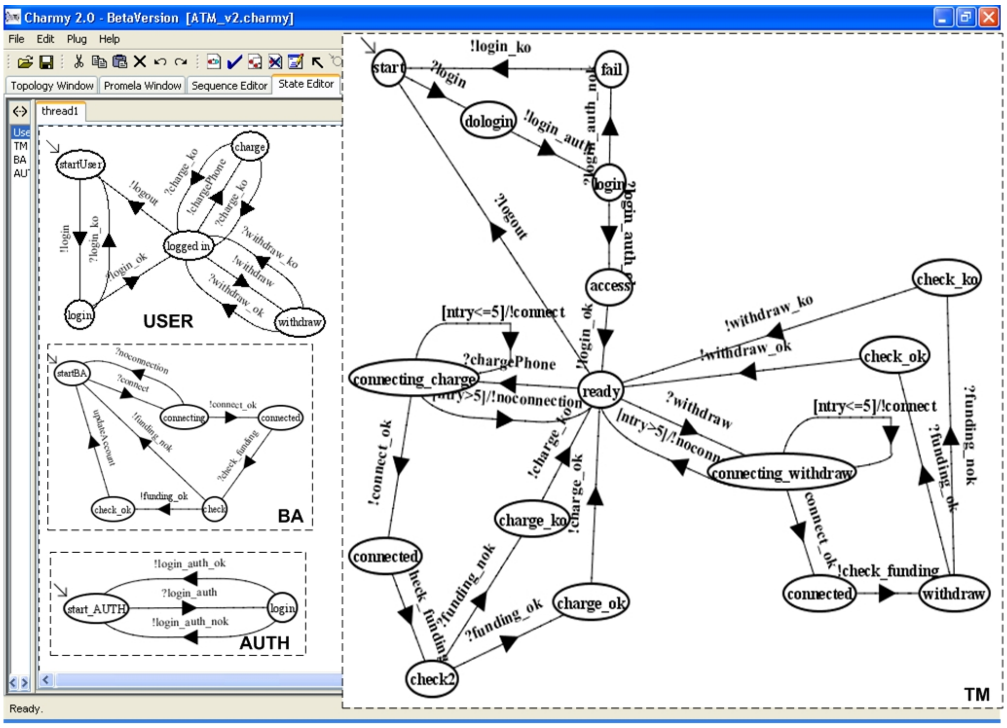


Figure 4. State diagrams of the ATM components

if three conditions hold: i) S$_i$ and S$_j$ have the same names, ii) they have the same incoming and outgoing transitions (same transition names), and iii) they have the same source and target states. Whenever these three conditions apply, the state is copied in STD$_{diff}$ and colored in green. If at least one condition applies (but not all of them), then the state is marked as changed and colored in yellow. If state S_i in STD$_1$ has not an equivalent state in STD$_2$, then S_i has been removed when transiting from STD$_1$ to STD$_2$. If, instead, S_i in STD$_2$ has not an equivalent state in STD$_1$, then S_i has been added when transiting from STD$_1$ to STD$_2$. Added states are colored in red, while removed ones are represented as dotted circles.

```
Procedure: stateNameComparison
input: STD_i, STD_i+1
output: STD_diff

Repeat
 s_i = getElementState(STD_i);
 markAsVisited(s_i,STD_i);
 if exists(s_i,STD_i+1)
   then markAsVisited(s_i,STD_i+1);
     if (getName(s_i,STD_i) !=
            getName(s_i,STD_i+1))
       then addAsChanged(s_i,STD_diff)
       else if (getSource(s_i,STD_i) =
                    getSource(s_i,STD_i+1) &
               (getTarget(s_i,STD_i) =
                    getTarget(s_i,STD_i+1))
            then addAsSame(s_i,STD_diff)
   else addAsRemoved(s_i,STD_diff)
Until notVisited(STD_i) =! empty
if notVisited(STD_i+1) =! empty
    then foreach(s_i,notVisited(STD_i+1))
        addAsNew(s_i,STD_diff)
```

After state name comparison, the algorithm proceeds with transition name similarity. STD$_1$ *ELEMENTMESSAGE* tags are traversed and compared with STD$_2$ transitions. In order to identify transition equivalence, two transitions T_i and T_j are the same if two conditions hold: i) the transition label is the same (*ACTIONS*, *CONDITIONS*, and *PARAMETERS* tags), and ii) the target and source states are the same (*SOURCE* and *TARGET* tags). A non modified transition is represented in green. In case only one condition holds, the transition has been modified and it is colored in yellow. Added transitions are colored in red, while removed ones are represented via dotted lines.

A version 3 of the ATM example has been produced (ATM_v3), where the User can also check her money balance, and the TM has a different connection management: first, a connection with the BA is established, then the withdraw, charge, or money balance services are activated according to the User request (while in the previous version, when a service is activated, it opens its own connection). By applying the model differencing algorithm to the TM_{diff} component, we obtained the diagram in Figure 6. In green, states and transitions preserved. In red, new states and transitions. In yellow, modified states and transitions.

By taking a closer look to the TM_{diff} diagram, we can make the following observations: *change_ok* and *change_ko* states are in green since they have same names, same incoming and outgoing transitions, and same source and target states in the two versions. *Connecting* is a modified state (i.e., in yellow) since, even if source and target states are unchanged, they do have modified incoming transitions and names. *Balance* is a new state (thus in red) since there is no state with the same name, and incoming and outgoing transitions, and source and target state differ. *?Connect_ok* is a non modified transition (i.e., in green) since both names and source and target states are the same. *?ChangePhone* is modified since the source state differs in the two versions. *!Moneyreport* is a new transition (thus in red) since there is no transition with such name in the first version, and there is no other transition with the same source and target states.

4.3 Test Case re-Selection

In *selective regression testing*, given P a software system, P′ its evolution, and T a test suite for P, $t \in T$ is selected to be re-executed in P′ (i.e., $t \in T'$) if there is the potential that it could produce different results on P′ that it did on P (following a *safe* definition). Dangerous regions are identified to denote portions of the system that, if traversed, may lead to a different behavior in P and P′ [17]. Whenever a test case $t \in T$ causes P to traverse a dangerous region, *t* traverses a modified code that may cause a different behavior in P and P′, and *t* must be re-run on T′.

Our approach identifies dangerous regions by model differencing. Any node/arc in the STD_{diff} graph which has been colored in red, yellow, or via dotted lines represent dangerous regions.

The test case re-selection algorithm presented in this section performs two different actions, thus implementing steps 1 and 3 in Section 2: i) it checks whether an existing test case $t \in T$ must be re-tested in T′ (regression test selection phase), and ii) it identifies new test cases t′′ necessary to cover added portions of the CBSA (new test case selection phase).

The regression test selection phase is performed by simulating each test case $t \in T$ in the STD_{diff} graph. Whenever the test case covers yellow or dotted states and transitions, then it needs to be re-run on P′. Whenever it covers red states or transitions, new test cases might be needed to test the added portion of the CBSA.

By taking into consideration the ATM example, and the seven test cases previously described, we notice that most of them need to be re-run. By solely focus on the TM component, test cases $t1$ and $t2$ do not need to be re-tested since they cover only green states and transitions (the upper left part of the TM_{diff} graph in Figure 6). $t3$ needs to be re-tested, since it traverses the *!connect* transition marked in red and the *connecting* state in yellow. Since any other test case $t4$ to $t7$ traverses the *!connect* transition and the *connecting* state, they need to be re-tested all. This is due to the fact that the TM component behavior has been conceptually restructured (connection first, than any of the service). Assuming the only change from version ATM_v2 to ATM_v3 would have been adding the check money balance feature, none of the seven test cases would have been re-tested.

The new test case selection phase is instead implemented by generating test cases covering the added states and transitions. In the ATM_v3 specification, the new states are *balance*, *waiting*, *report*, and *check1*, while the new transitions are *!connect*, *?moneybalance*, *!moneyreport*, *?report*, and *print*. By re-applying the same coverage criteria applied in the initial graph (i.e., any edge in the STD graph), *!connect* is already covered by the test cases $t3$-$t7$, new test cases are needed for the other transitions. Two test cases are needed for covering all the new transitions: $t8'$ – the User logins, get connected, asks for money balance, and obtains a printed statement, and $t9'$ – the User logins, get connected, asks for money balance, but does not receive a printed statement since there is no paper in the ATM machine. In summary, seven test cases need to be run on ATM_v3: $t3$-$t7$, $t8'$, and $t9'$.

5 Tool support

The regression testing method has been implemented inside the CHARMY framework through three main modules.

The CHARMY standard editor plugin which allows the model-based specification of the CBSA under analysis. It allows the specification of a CBSA through a topology editor and a behavioral diagram (as summarized in Section 4 and shown in Figures 3 and 4).

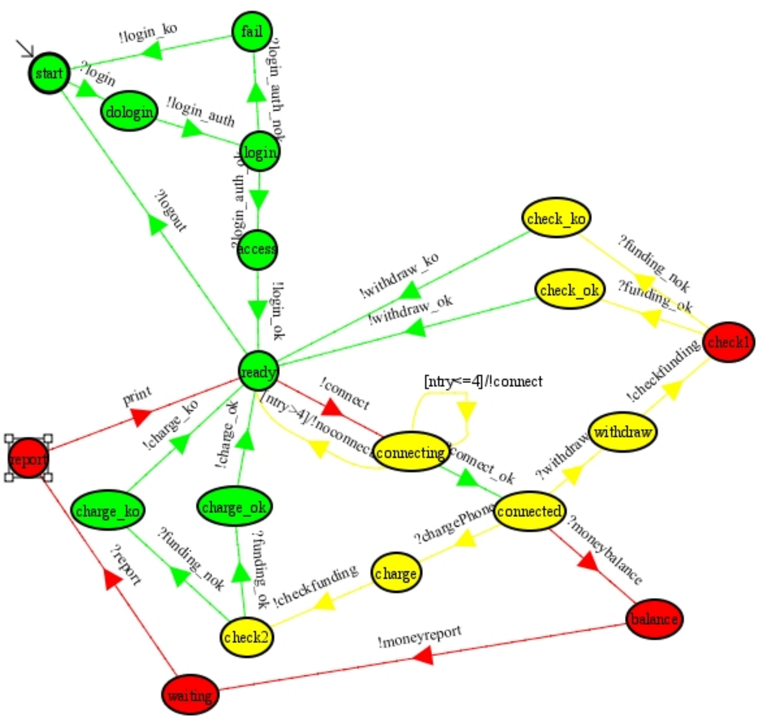


Figure 6. Modified State diagram for the TM component

The CHARMY architectural testing plugin, recently developed, which allows the test cases selection from the architectural specification. This module implements the approach in [14] by generating a global automaton out of the component state diagrams, by applying an observation so to focus on relevant (with respect to a test purpose) portions of the graph, and by implementing various coverage criteria [24, 19] (all edges, all components, all states, McCabe). The CADP (Construction and Analysis of Distributed Processes) [2] tool features have been utilized in many steps in the test case selection process, and integrated with CHARMY.

The CHARMY regression testing plugin, currently in its beta version, which implements the model differencing and test re-selection algorithms described in this paper. The produced output is a colored graph highlighting model differences (as the one in Figure 6) and a set of re-selected test cases.

6 Initial Considerations

This initial work on model differencing-based regression testing seems to be very promising, due to the current interest on model-based specifications of software architectures (see e.g., [3]) and to the ongoing work on model differencing techniques.

However, many issues still need to be resolved. First of all, it is important to understand how much the algorithms are useful and effective for selective regression testing. According to [22] five are the main categories which should be taken into account when evaluating selective retest strategies: inclusiveness, precision, efficiency, generality, and accountability. More study is necessary in order to be able to position our approach according to those parameters. However, initial study let us believe that as far as concern *precision* (which measures the ability of a RT technique to omit tests that are non-modification-revealing) the proposed technique is *safe* (i.e., it re-selects any relevant test case) even if its *inclusiveness* (which measures the extent to which a selective retest strategy chooses modification revealing tests) can be improved by making the algorithms considering finer grained architectural changes. This approach considers only the impact of architectural changes on the test case re-selection process, while not taking into account code changes. Moveover, the basic assumption of our approach is that it is possible to trace architectural components to real components by focussing on the concrete architecture of a CBS (called CBSA).

7 Conclusions and future Work

Regression testing can be systematically applied at the software architecture level in order to anticipate the testing phase as soon as possible in the software design stage, while reducing the cost of retesting modified systems. While previous effort has investigated what regression testing at the architectural level means [9] and how it can be conceptually implemented in a testing and regression testing framework [15], this paper has shown how the regression test selection and the new test case selection phases can be implemented through model differencing. This paper has outlined two algorithms: one for model differencing of CHARMY state machines, one for test re-selection based on the model differencing results. Initial results have been shown in the context of the ATM running example.

In future work we do plan to improve tool support and to apply our approach to a complex example. From a conceptual view point, we want to understand how this approach can be applied when subsequent evolutions of the CBSA apply (linear derivation), or when the same version evolves according to different evolution branches (branching derivation).

Acknowledgment

The author of this paper wish to thank Linda Corsetti who implemented part of the model differencing regression testing approach and the anonymous reviewers who provided relevant comments and suggestions on how to improve it.

References

[1] Comparing and merging UML models in IBM Rational Software Architect. http://www-128.ibm.com/developerworks/rational/library/05/712_comp/.

[2] Construction and Analysis of Distributed Processes (CADP) project homepage. http://www.inrialpes.fr/vasy/cadp/.

[3] The SAE Architecture Analysis and Design Language (AADL). http://www.aadl.info/.

[4] CHARMY Project. Charmy Web Site. http://www.di.univaq.it/charmy, 2004.

[5] A. Bertolino and P. Inverardi. Architecture-based Software Testing. In *Proc. ISAW96*, October 1996.

[6] A. Cicchetti, D. D. Ruscio, and A. Pierantonio. A DSML for Model Differences. In *European Workshop on Composition of Model Transformations, CMT 2006*, 2006.

[7] N. Eickelman and D. Richardson. What Makes One Software Architecture More Testable Than Another? In *In Proc.* ISAW-2, October 1996.

[8] T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel. An Empirical Study of Regression Test Selection Techniques. In *Proc. of the 20th Int. Conf. on Software Engineering (ICSE'98)*, pages 188–197, Japan, April 1998.

[9] M. J. Harrold. Architecture-Based Regression Testing of Evolving Systems. In *Proc. Int. Workshop on the Role of Software Architecture in Testing and Analysis - ROSATEA*, pages 73–77, July 1998.

[10] M. J. Harrold. Testing: A Roadmap. In A. Finkelstein, editor, *ACM ICSE 2000, The Future of Software Engineering*, pages 61–72, 2000.

[11] Z. Jin and J. Offutt. Deriving tests from software architectures. In *ISSRE*, pages 308–313, 2001.

[12] U. Kelter, J. Wehren, and J. Niere. A Generic Difference Algorithm for UML Models. In *Proceedings of the SE 2005*, Essen, Germany, March 2005.

[13] P. Kruchten. Architectural Blueprints - The "4+1" View Model of Software Architecture. *IEEE Software*, 12(6):42–50, November 1995.

[14] H. Muccini, A. Bertolino, and P. Inverardi. Using Software Architecture for Code Testing. *IEEE Trans. on Software Engineering*, 30(3):160–171, March 2003.

[15] H. Muccini, M. Dias, and D. J. Richardson. Software Architecture-based Regression Testing. *Int. Journal of Systems and Software, special issue on Architecting Dependable Systems*, To appear 2006.

[16] L. Naslavsky, D. Richardson, and H. Ziv. Scenario-based and State Machine-based Testing: An Evaluation of Automated Approaches. Technical report, University of California, Irvine - ISR TR UCI-ISR-06-13, 2006.

[17] A. Orso, N. Shi, and M. J. Harrold. Scaling regression testing to large software systems. In *Proceedings of the 12th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE 2004)*, pages 241–252, 2004.

[18] P. Pelliccione. *CHARMY: A framework for Software Architecture Specification and Analysis*. PhD thesis, Computer Science Dept., University of L'Aquila, May 2005.

[19] D. J. Richardson, J. Stafford, and A. L. Wolf. A Formal Approach to Architecture-based Software Testing. Technical report, University of California, Irvine, 1998.

[20] D. J. Richardson and A. L. Wolf. Software Testing at the Architectural Level. In *ISAW-2, in Joint Proc. of the ACM SIGSOFT '96 Workshops*, pages 68–71, 1996.

[21] D. Rosenblum. Challenges in Exploiting Architectural Models for Software Testing. In *Proc. Int. Workshop on the Role of Software Architecture in Testing and Analysis - ROSATEA*, July 1998.

[22] G. Rothermel and M. J. Harrold. A Framework for Evaluating Regression Test Selection Techniques. In *In Proc. 16th Int. Conference on Software Engineering, ICSE 1994*, pages 201–210, Sorrento, Italy, May 1994.

[23] J. A. Stafford and A. L. Wolf. Architecture-level dependence analysis in support of software maintenance. In *ISAW '98: Proceedings of the third international workshop on Software architecture*, pages 129–132, New York, NY, USA, 1998.

[24] A. H. Watson and T. J. McCabe. Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric. *NIST Special Publication 500-235*, August 1996.

[25] Z. Xing and E. Stroulia. Differencing logical UML models. *Special issue of Automated Software Engineering Journal. Selected papers from ASE 2005. To appear.*