CRIM

CRIM — Documentation/Communications

Proceedings of the 22nd IFIP International Conference on Testing Software and Systems: Short Papers

Editors
Alexandre Petrenko
Adenilso Simão
José Carlos Maldonado

October, 2010

ISBN-13: 978-2-89522-136-4

Financial Partner:

Développement économique, Innovation et Exportation Québec

Preface

Testing has steadily become more and more important within the development of various software and systems, motivating an increasing amount of research, trying to solve both new challenges imposed by the advancement in other areas of computer science and long-standing problems. Testing has evolved during the last decades from an ad-hoc and under-exposed area of systems development to an important and active research area. The 22nd International Conference on Testing Software and Systems (ICTSS) is the merge of two traditional and important events which have served the testing community as an important venue for discussing advancements in the area. Those events, namely, TestCom the IFIP TC6/WG6.1 International Conference on Testing of Communicating Systems, and Fates International Workshop on Formal Approaches to Testing of Software, together form a large event on testing, validation, and specification of software and systems. They have a long history. TestCom Testing of Communicating Systems is an IFIP-sponsored series of international conferences, previously also called International Workshop on Protocol Test Systems (IWPTS) or International Workshop on Testing of Communicating Systems (IWTCS). It is devoted to testing of communicating systems, including testing of communication protocols, services, distributed platforms, and middleware. The previous events were held in Vancouver, Canada (1988); Berlin, Germany (1989); McLean, USA (1990); Leidschendam, The Netherlands (1991); Montreal, Canada (1992); Pau, France (1993); Tokyo, Japan (1994); Evry, France (1995); Darmstadt, Germany (1996); Cheju Island, South Korea (1997); Tomsk, Russia (1998); Budapest, Hungary (1999); Ottawa, Canada (2000); Berlin, Germany (2002); Sophia Antipolis, France (2003); Oxford, UK (2004); Montreal, Canada (2005); and New York, USA (2006). Fates Formal Approaches to Testing of Software is a series of workshops devoted to the use of formal methods in software testing. Previous events were held in Aalborg, Denmark (2001); Brno, Czech Republic (2002); Montreal, Canada (2003); Linz, Austria (2004); Edinburgh, UK (2005); and Seattle, USA (2006). From 2007 on, TestCom and Fates have been jointly held in Tallinn, Estonia (2007), Tokyo, Japan (2008) and Eindhoven, The Netherlands (2009). The objective of ICTSS 2010 was to be a forum for researchers from academia as well as industry, developers, and testers to present, discuss, and learn about new approaches, theories, methods and tools in the field of testing software and systems.

The accepted full papers of the conference were published by Springer as the LNCS volume 6435 "Testing Software and Systems". These proceedings contain accepted short papers of ICTSS 2010. It is published by CRIM, Canada, which is one of the organizers of this conference.

October 2010

Alexandre Petrenko
Adenilso Simao
Jose Carlos Maldonado

Conference Organization

Program Chairs

Alexandre Petrenko (CRIM, Canada)
Adenilso Simao (University of Sao Paulo, Brazil)
Jose Carlos Maldonado (University of Sao Paulo, Brazil)

Steering Committee

Paul Baker (Motorola, UK)
Ana R. Cavalli (Telecom SudParis, France)
John Derrick (University of Sheffield, UK) (Chair)
Wolfgang Grieskamp (Microsoft Research, USA)
Roland Groz (Grenoble Institute of Technology, France)
Toru Hasegawa (KDDI R&D Labs., Japan)
Manuel Nunez (University Complutense de Madrid, Spain)
Alexandre Petrenko (CRIM, Canada)
Jan Tretmans (Embedded Systems Institute, The Netherlands)
Andreas Ulrich (Siemens AG, Germany)
Margus Veanes (Microsoft Research, USA)

Program Committee

Paul Baker (Motorola, UK)
Antonia Bertolino (ISTI-CNR, Italy)
Roberto S. Bigonha (Federal University of Minas Gerais, Brazil)
Gregor v. Bochmann (University of Ottawa, Canada)
Ana R. Cavalli (Telecom SudParis, France)
John Derrick (University of Sheffield, UK)
Sarolta Dibuz (Ericsson, Hungary)
Khaled El-Fakih (American University of Sharjah, UAE)
Gordon Fraser (Saarland University, Germany)
Wolfgang Grieskamp (Microsoft Research, USA)
Roland Groz (Grenoble Institute of Technology, France)
Toru Hasegawa (KDDI R&D Labs., Japan)
Klaus Havelund (Jet Propulsion Laboratory, USA)
Rob Hierons (Brunel University, UK)
Teruo Higashino (Osaka University, Japan)
Dieter Hogrefe (University of Gottingen, Germany)
Antti Huima (Conformiq Inc., USA)
Thierry Jeron (IRISA Rennes, France)
Ferhat Khendek (Concordia University, Canada)
Myungchul Kim (ICU, Korea)
Hartmut Konig (BTU Cottbus, Germany)

Victor V. Kuliamin (ISP RAS, Russia)
David Lee (Ohio State University, USA)
Bruno Legeard (Smartesting, France)
Patricia Machado (Federal University of Campina Grande, Brazil)
Giulio Maggiore (Telecom Italia Mobile, Italy)
Jose Carlos Maldonado (University of Sao Paulo, Brazil)
Eliane Martins (University of Campinas, Brazil)
Ana Cristina de Melo (University of Sao Paulo, Brazil)
Brian Nielsen (University of Aalborg, Denmark)
Daltro Jose Nunes (Federal University of Rio Grande do Sul, Brazil)
Doron Peled (University of Bar-Ilan, Israel)
Alexandre Petrenko (CRIM, Canada)
S Ramesh (General Motors India Science Lab, India)
Augusto Sampaio (Federal University of Pernambuco, Brazil)
Ina Schieferdecker (Fraunhofer FOKUS, Germany)
Adenilso Simao (University of Sao Paulo, Brazil)
Kenji Suzuki (University of Electro-Communications, Japan)
Jan Tretmans (Embedded Systems Institute, The Netherlands)
Andreas Ulrich (Siemens AG, Germany)
Hasan Ural (University of Ottawa, Canada)
M. Umit Uyar (City University of New York, USA)
Margus Veanes (Microsoft Research, USA)
Cesar Viho (IRISA Rennes, France)
Carsten Weise (RWTH Aachen, Germany)
Burkhart Wolff (University of Paris-Sud, France)
Nina Yevtushenko (Tomsk State University, Russia)
Xia Yin (Tsinghua University, China)

Local Chair

Marcel Oliveira (Federal University of Rio Grande do Norte, Brazil)

Local Organization

Thais Batista (Federal University of Rio Grande do Norte, Brazil)
David Deharbe (Federal University of Rio Grande do Norte, Brazil)

Table of Contents

Formal conformance verification

Igor Burdonov[1], Alexander Kosachev[1],

[1] Institute for System Programming of the Russian Academy of Sciences,
A. Solzhenitsyna st. 25, 109004 Moscow, Russia
{igor, kos}@ispras.ru

Abstract. In this paper, we propose a method for verification the so-called ***saco***-relation between an implementation and the specification LTSs. We show that this verification can be finite and complete if the number of states and actions of LTSs is finite.

Keywords: LTS model, conformance relation, formal verification.

1 Introduction

In formal model based conformance verification, one usually assumes that the specification and a system under test (SUT) are represented by the same formal models and there is a binary relation that has to be checked via the verification process. In this paper, we limit ourselves with so-called functional restrictions which describe how a system should interact with its environment; these restrictions are described by the specification. A SUT is conforming if its interaction satisfies the same external restrictions.

In various application domains the functional behavior (or the specification) of a system can be described by the LTS model. In this paper, we assume that we should compare two LTS models, one of which describes the reference behavior while another one describes the behavior of a SUT. We introduce a conformance relation between LTSs and show how this conformance relation can be verified.

2 Conformance relation and how to check it

2.1 Interaction and its safety

The conformance relation is based on the interaction model. We can only observe the behavior that is induced by the environment and can be observed by the environment. Such interaction can be modeled by a testing machine [1-6]. The testing machine is a black box where an implementation (SUT) is embedded. The environment is modeled by an operator who presses buttons allowing an implementation to execute some

actions. Observations can be of two types: an implementation executes an allowed action or an implementation refuses to execute any allowed action.

We underline that not a single action but a set of actions is allowed when pressing a button. For example, when talking about reactive systems the set of actions is partitioned into two sets of inputs and outputs. Each input action is applied to an implementation by pressing a button that corresponds to this input. However, when expecting the reply of an implementation a button is pressed that allows to accept any output action.

Each button has its own set of permissible actions. An action can be observed if the action is allowed by a pressed button and an implementation can execute the action. If an implementation cannot execute any action allowed by a pressed button then a refusal is observed. After observing execution of an action or after observing the refusal all external actions are forbidden before another button is pressed.

Thus, interaction depends on sets of actions associated with buttons as well as on the set of buttons for which a refusal can be observed. For the sake of simplicity, in this paper, we assume that all refusals are observable. Correspondingly, the interaction is completely described by a set L of external actions and a collection R of sets of actions associated with buttons. We also assume that $\cup R = L$ and refer to such an interaction as to R-interaction.

An implementation can often execute not only external actions but also the internal (non-observable) action τ. Such actions are always allowed. As usual, we assume that a finite sequence of internal actions needs finite time to be executed while an infinite sequence of such actions can be executed infinitely long. An infinite sequence of τ-actions is called divergence and is denoted by Δ. Divergence can induce a problem, since when pressing a button an operator does not know how long he has to wait for a response or internal actions will be executed forever. Therefore, an operator is deadlocked as he cannot continue the interaction and cannot stop it.

We also consider a special action that cannot also be controlled by buttons; this actions called destruction is denoted by γ. The action γ models undesirable system behavior including the system destruction that is not allowed through ordinary interaction. For example, in defense devices such an action can correspond to self-destruction. Moreover, the destruction can describe the non-specified behavior in partial specifications. When people do not consider a destruction they motivate this that an implementation should only check if parameters are correct when calling an implementation [7,8]. If parameters of a message are incorrect then the implementation should ignore a message or should mention about an error. Such requirement is understandable when an implementation is for “a common use” and should be foolproof. However, when interacting with internal components or subsystems to which an access is strongly limited such checking seems to be superfluous. If the parameter structure is complex and the correctness conditions are not trivial such checking adds unnecessary complexity when designing a system and the system performance is also adversely affected. There is an alternative to use the strong specification of calling operators [9]. For instance, a call for deallocation memory that was not got earlier via memory request is incorrect as it violates preconditions. Correspondingly we should check not how components response to incorrect requests of other components but whether those requests are correct. In other words, since we would like to assure the implementation correctness (not its

environment) we are not interested in the implementation behavior when a request is incorrect. Specification destruction describes situations when an implementation is allowed to have any behavior including the real destruction.

Destruction semantics assumes that the destruction cannot happen if the environment behavior is correctly implemented. The interaction is safe if the destruction cannot occur and buttons are pressed only there isn't divergence in an implementation.

2.2 LTS and LTS traces

LTS is a tuple $\boldsymbol{S} = LTS(V_S,L,E_S,s_0)$ where V_S is the non-empty set of states with the initial state s_0, L is the non-empty set of external actions, $E_S \subseteq V_S \times (L \cup \{\tau,\gamma\}) \times V_S$ is the set of transitions. As usual, a transition from state s to state s' via action z is denoted $s \text{—} z \rightarrow s'$ while $s \text{—} z \rightarrow$ denotes that there is a transition from state s via action z, $s \text{—} z \rightarrow =_{\mathbf{def}} \exists s'(s \text{—} z \rightarrow s')$. A sequence of sequential transitions is a *path* of LTS.

A step of LTS functioning in a testing machine is an execution of a single transition that is allowed by a current button and is specified at a current state (τ- and γ-transitions are always allowed). If there are several such transitions then only one of them can be executed.

State s is *divergent* ($s\uparrow$) if s is the initial state of an infinite *path* of τ-transitions (in particular, of a τ-cycle); otherwise, state s is *convergent* ($s\downarrow$). State is *stable* if there are no τ- or γ-transitions at this state. Refusal $P \in R$ occurs at a stable state if at this state there are no transitions under actions of the set P.

At each stable state we add a virtual loop $s \text{—} P \rightarrow s$ for each possible refusal P, while at each divergent state Δ-transitions $s \text{—} \Delta \rightarrow$ are added. In the obtained LTS we consider only *paths* which are not prolonged after Δ- and γ-transitions. A *trace* is a sequence of action labels of path transitions without symbol τ. If a path with trace σ has the initial state s and the final state s' then we denote this by $s \Rightarrow \sigma \Rightarrow s'$ while $s \Rightarrow \sigma \Rightarrow$ denotes that there is a trace σ with the initial state s, $s \Rightarrow \sigma \Rightarrow =_{\mathbf{def}} \exists s' \, (s \Rightarrow \sigma \Rightarrow s')$. The set $T(s) = \{\sigma \mid s \Rightarrow \sigma \Rightarrow\}$ is the set of traces at state s.

2.3 Safety hypothesis and safe conformance

We now define the relation "a button is safe after a trace": button P is *safe after trace* σ if there is no divergence after the trace and the destruction cannot occur after an action allowed by the button P. Only safe buttons are pressed through the safe interaction. Formally, the button P is safe:

At state s: P ***safe*** $s =_{\mathbf{def}} s\downarrow \;\&\; \neg s \Rightarrow \langle\gamma\rangle \Rightarrow \;\&\; \forall z \in P \; \neg s \Rightarrow \langle z,\gamma\rangle \Rightarrow$;

For the set S of state: P ***safe*** $S =_{\mathbf{def}} \forall s \in S \; P$ ***safe*** s;

After trace σ with the starting state s: P ***safe*** s ***after*** $\sigma =_{\mathbf{def}} P$ ***safe*** (s ***after*** σ),

where s ***after*** $\sigma =_{\mathbf{def}} \{s` \mid s \Rightarrow \sigma \Rightarrow s`\}$.

The button safety implies the safety of actions and refusals. Refusal P is *safe* if the button P is safe. Action z is *safe* if this action is allowed by some safe button P, i.e., $z \in P$. A trace with the initial state s is safe if 1) $\neg(s \Rightarrow \langle\gamma\rangle \Rightarrow)$, 2) and each trace action

is safe after the corresponding trace prefix. The set of all safe traces at state s is denoted by $\boldsymbol{Safe}(s)$. By default, the initial state of all traces and paths is the initial state of the LTS.

The safety requirement results in the set of safe implementations. This set is determined by the *safety hypothesis*: implementation $\boldsymbol{I}$ is *safe* for the specification $\boldsymbol{S}$ if 1) the implementation does not contain a trace $\langle\gamma\rangle$ if there is no such trace at the initial state of the specification 2) after each common safe trace of an implementation and the specification any button that is safe in the specification is safe in the implementation after this trace:

$\boldsymbol{I}$ ***safe for*** $\boldsymbol{S} =_{\mathbf{def}} (\gamma \notin T(s_0) \Rightarrow \gamma \notin \boldsymbol{T}(i_0))$ & $\forall \sigma \in \boldsymbol{Safe}(s_0) \cap T(i_0)$ $\forall P \in R$

(P ***safe*** s_0 ***after*** $\sigma \Rightarrow P$ ***safe*** i_0 ***after*** σ).

An implementation $\boldsymbol{I}$ is *conforming* to the specification $\boldsymbol{S}$ if $\boldsymbol{I}$ is safe and the following *conformance condition* holds: any observation that is possible in the implementation after pressing a safe button is allowed by the specification.

$\boldsymbol{I}$ ***saco*** $\boldsymbol{S} =_{\mathbf{def}} \boldsymbol{I}$ ***safe for*** $\boldsymbol{S}$ &

$\forall \sigma \in \boldsymbol{Safe}(s_0) \cap T(i_0)$ $\forall P$ ***safe*** s_0 ***after*** σ ($\boldsymbol{obs}(i_0$ ***after*** $\sigma, P) \subseteq \boldsymbol{obs}(s_0$ ***after*** $\sigma, P)$),

where $\boldsymbol{obs}(M,P) =_{\mathbf{def}} \{u \in P \cup \{P\} \mid \exists m \in M$ & $m \Rightarrow \langle u \rangle \Rightarrow\}$ is the set of observations which are possible at states of the set M when pressing the button P.

When the safety hypothesis cannot be checked the hypothesis becomes a precondition of testing; the testing objective then is to check the conformance relation. Both conditions can be checked when formal verification is used.

3 Verification technique

We first establish restrictions for the interaction model, implementation and specification which allow finite conformance verification and which are considered in this paper. We assume that the set L of actions is finite, thus, the number of the sets R is finite, correspondingly, a set corresponded to each button and the set $L \cup R$ of observations also are finite; moreover, the sets of states of an implementation and the specification are finite, i.e., their sets of transitions are finite.

The idea behind our approach is as follows. Consider all the states of an implementation which are reachable by traces that are safe in the specification: $\cup\{i_0$ ***after*** $\sigma \mid \sigma \in \boldsymbol{Safe}(s_0)\}$. For each such state consider a collection $S(i)$ of subsets of states of the specification: $S(i) = \{s_0$ ***after*** $\sigma \mid \sigma \in \boldsymbol{Safe}(s_0)$ & $i \in (i_0$ ***after*** $\sigma)\}$. This collection has subsets of states of the specification after all traces which are safe in the specification, are traces of the implementation and have the final state i. When verifying we will construct such collections $S(i)$ step by step adding to them sets s_0 ***after*** σ. At the beginning we check the condition $\langle\gamma\rangle \notin T(s_0) \Rightarrow \langle\gamma\rangle \notin T(i_0)$ as a part of the safety hypothesis. If this condition holds then at each time instance at each state i another part of the safety hypothesis and of the conformance relation is checked: $\forall P \in R$ $\forall S \in S(i)$ (P ***safe*** $S \Rightarrow P$ ***safe*** i & $\boldsymbol{obs}(\{i\},P) \subseteq \boldsymbol{obs}(S,P)$). If these conditions do not hold then a fault is claimed. If all the collections $S(i)$ are completely constructed then the verification is completed with the verdict "OK".

We first construct intermediate data structures for the specification and implementation. Specification data structures do not depend on an implementation

and are used without modification when verifying any implementation using the same *R*-interaction. Correspondingly, implementation data structures do not depend on the specification and can be used for checking its conformance w.r.t. any specification when using the same *R*-interaction. Correspondingly, we would not take into account such data structures when estimating the verification algorithm complexity.

For the specification, we consider a collection of subsets of final states of safe traces: $\boldsymbol{safeder}(s_0) = \{s_0 \textbf{ after } \sigma | \sigma \in \boldsymbol{Safe}(s_0)\}$. For each subset $S \in \boldsymbol{safeder}(s_0)$, the set $A(S) = (\cup\{P \cup \{P\} | P \textbf{ safe } S\}) \cup \{\tau\}$ contains all safe observations (actions and refusals) and symbol τ while the set $B(S,u) = \cup\{s \textbf{ after } \langle u \rangle | s \in S\}$ contains each state s' such that there is a transition (s,u,s'), $s \in S$ in the specification; we also assign $B(S,\tau) = S$. Apparently, $B(S,u) = \varnothing$ if there are no such transitions.

In fact, when proceeding in the way discussed above we derive the observable form of the specification LTS; states of the observable form are sets of states $S = s_0 \textbf{ after } \sigma$, where $\sigma \in \boldsymbol{Safe}(s_0)$, a transition $S \text{—} u \rightarrow S'$ means that $S' = B(S,u)$ and $B(S,u) \neq \varnothing$.

For an implementation, we consider a set of states which are reachable via safe traces (in the implementation): $\cup\boldsymbol{safeder}(i_0)$. For each state $i \in \cup\boldsymbol{safeder}(i_0)$ we define the following sets:

$C(i) = \langle i \text{—} u \rightarrow i` | u = \tau \vee \exists P \textbf{ safe } i\ u \in P \rangle$ is a set of all safe transitions at state i;

$D(i) = \langle i \text{—} u \rightarrow i` | \forall P \textbf{ safe } i\ u \notin P \rangle$ is a set of transitions at state i which are not safe.

The safe hypothesis for an implementation is equivalent to the condition

$\forall i \text{—} u \rightarrow i` \in D(i)\ \forall S \in S(i)\ u \notin A(S)$

while the conformance relation is equivalent to the condition

$\forall i \text{—} u \rightarrow i` \in C(i)\ \forall S \in S(i)\ u \in A(S) \Rightarrow B(S,u) \neq \varnothing$.

A proposed algorithm is based on deriving sets $S(i)$ step by step while checking safety and conformance at each step. A special intermediate list W of pairs $(S, i \text{—} u \rightarrow i`)$ is constructed where $S \in S(i)$ and $i \text{—} u \rightarrow i` \in C(i) \cup D(i)$. At the beginning it holds that $S(i_0) = \{S_0\}$ where $S_0 = \{s_0 \textbf{ after } \langle\rangle\}$ is a set of states of the specification after the empty trace and W has each pair $(S_0, i_0 \text{—} u \rightarrow i`)$ where $i_0 \text{—} u \rightarrow i` \in C(i_0) \cup D(i_0)$.

An algorithm step is as follows. If the list W is empty then the algorithm stops with the verdict "conforming". Otherwise, the head item $(S, i \text{—} u \rightarrow i`)$ of the list $W$ is selected and is deleted from the list. For the selected item, first, the safety hypothesis is checked. If $i \text{—} u \rightarrow i` \in D(i)$ & $u \in A(S)$ then an implementation is claimed as a faulty implementation and the algorithm stops. If there is no fault then the conformance is checked. If $i \text{—} u \rightarrow i` \in D(i)$ or $i \text{—} u \rightarrow i` \in C(i)$ & $u \notin A(\mathrm{S})$ then the next step of the algorithm is performed. If $i \text{—} u \rightarrow i` \in C(i)$ & $u \in A(\mathrm{S})$ & $B(\mathrm{S},u) = \varnothing$ then an implementation is claimed as a faulty implementation and the algorithm stops. If there is no fault and $B(\mathrm{S},u) \in S(i`)$ then the next step of the algorithm is performed. Otherwise, the set $B(\mathrm{S},u)$ is added to $S(i`)$ while each pair $(B(\mathrm{S},u), i` \text{—} u` \rightarrow i``)$ where $i` \text{—} u` \rightarrow i`` \in C(i`) \cup D(i`)$ is added to the list W, and the next step of the algorithm is performed.

It is easy to show that an implementation is claimed as a conforming implementation if and only if the implementation conforms to the specification.

We also notice that the algorithm is easy for parallel programming, since algorithm steps corresponded to different items of the list W can be performed in parallel; if

there is no fault then the algorithm stops when there are no steps to be performed and the list W is empty.

We now estimate the algorithm complexity in the worst case that is in the case when parallel programming is useless. Since each pair $(S, i \text{—} u \rightarrow i')$ can appear in the list W at most once, the number of algorithm steps is equal to $O(m \cdot 2^n)$ where $m=|E_I|$ is a number of implementation transitions, and $n = |V_S|$ is the number of states of the specification. At each step we can assume that time needed for all actions such as checking whether an item belongs to a given set, extracting items of the set B and adding an item to $S(i')$, is a proper constant, all observations, all implementation states and all subsets of the set of specification states are hashed by integers, A and S are two-dimensional Boolean arrays, C and D are three-dimensional Boolean arrays and B is a two-dimensional array of numbers of subsets of the set of specification states. The exception should be made for adding pairs $(B(S,u), i' \text{—} u \rightarrow i'')$ to the list W where $i' \text{—} u \rightarrow i'' \in C(i') \cup D(i')$. Summarizing the above we obtain $O(m \cdot 2^n)$ pairs. If sets of transitions $C(i')$ and $D(i')$ are additionally represented as lists for every state i', this operator needs $O(m \cdot 2^n)$ time units to be executed. Thus, the resulting estimation equals $O(m \cdot 2^n)$. Coefficient 2^n is the maximal number of states of the observable form $(2^n - 1)$ of the LTS specification. If the observable form has the same number of states as the initial specification that happens when the specification is deterministic, i.e., has no τ transitions and at each state there is at most one transitions for each action, then this coefficient equals n. Intuitively seems that the algorithm complexity on average is much less especially when using parallel programming and we are intended to study this upper bound more rigorously.

References

1. Bourdonov, I.B., Kossatchev, A.S., and Kuliamin, V.V.: Formalization of Test Experiments. Programming and Computer Software, 2007, Vol. 33, No. 5, pp.239-260
2. Bourdonov, I.B., Kossatchev, A.S., and Kuliamin, V.V.: Conformance theory for the systems with input refusals and destruction . Moscow, Nauka, 2008 (in Russian) http://panda.ispras.ru/~RedVerst/RedVerst/Publications/TR-03-2006.pdf
3. Bourdonov, I.B.: Conformance theory for functional formal model based testing of software systems . D-r Thesis, Moscow, ISP RAS, 2008 (in Russian) http://panda.ispras.ru/~RedVerst/RedVerst/Publications/TR-01-2007.pdf
4. van Glabbeek, R.J.: The linear time - branching time spectrum. CONCUR'90 (J.C.M. Baeten and J.W. Klop, ed.), LNCS 458, Springer-Verlag, 1990, pp 278–297.
5. van Glabbeek, R.J.: The linear time - branching time spectrum II; the semantics of sequential processes with silent moves. CONCUR'93 (E. Best, ed.), LNCS 715, Springer-Verlag, 1993, pp. 66–81
6. Milner, R.: Modal characterization of observable machine behaviour. CAAP 81 (G. Astesiano and C. Bohm, ed.), LNCS 112, Springer-Verlag, 1981, pp. 25–34
7. Tretmans, J.: Test Generation with Inputs, Outputs and Repetitive Quiescence. Software-Concepts and Tools, Vol. 17, Issue 3, 1996.
8. Heerink, L.: Ins and Outs in Refusal Testing. PhD thesis, University of Twente, Enschede, The Netherlands, 1998.
9. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM, 12(10), October 1969.

Composability Test of BOM based models using Petri Nets

Imran Mahmood[1], Rassul Ayani[1], Vladimir Vlassov[1], and Farshad Moradi[2]

[1]Royal Institute of Technology (KTH), Stockholm, Sweden
[2]Swedish Defense Research Agency (FOI), Stockholm, Sweden

Abstract. Reusability is a widely used concept which has recently received renewed attention to meet the challenge of reducing cost and time of simulation development. An approach to achieve effective reusability is through composition of predefined components which is promising but a daunting challenge in the research community. Base Object Model (BOM) is a component-based standard designed to support reusability and composability in distributed simulation community. BOM provides good model representation for component reuse however this framework lacks capability to express semantic and behavioral matching at the conceptual level. Furthermore there is a need for a technique to test the correctness of BOM-compositions in terms of structure and behavior. In this paper we discuss verification of BOM based model and test its suitability for the intended purpose and objectives. We suggest a technique through which the composed model can automatically be transformed into a single Petri Net (PN) model and thus can further be verified using different existing PN analysis tools. We further motivate our approach by suggesting a deadlock detection technique as an example, and provide a case study to clarify our approach.

Keywords: Verification, Model Based testing, Composability, Petri Nets

1 Introduction

Software reuse is the process of creating dynamic systems from existing components instead of building them from scratch. Reusability has recently gained renewed interest as an effort to minimize the cost and time associated with the development process. Composability is one of the effective means to achieve reusability. The concept of Composability was pioneered by Mike Petty in his theory of composability in Modeling and Simulation (M&S) community[6], according to which, "Composability is the capability to select and assemble simulation components in various combinations into simulation systems to satisfy specific user requirements". Composability is divided into syntactic and semantic levels. Syntactic composability means that the components can be connected to each other. Semantic composability refers to the fact that the coupling of components is considered meaningful, computationally valid and conforms to the intended objectives of the simulation. Semantic composability is broken down into two

sub-levels namely Static Semantic, which means that the components have same understanding of the concepts and the relations between them, and Dynamic Semantic that deals with the behavioral correctness of the composition[3]. There are some other levels of composability such as Pragmatic composability which we do not consider in this paper.

BOM (Base Object Model) represents an integrated framework that posses the ability to rapidly compose simulation components. It provides a foundation to define and characterize these components at a conceptual level. BOM encapsulates information needed to formally represent a simulation component. BOM is a SISO standard and encapsulates information needed to describe a simulation component. BOM was introduced as a conceptual modeling framework for HLA (High Level Architecture) which is an IEEE standard for distributed simulation. State-machine, which is an essential part of BOM provides means to formalize the change in the state of an entity with respect to its corresponding actions, thus in a way it depicts the abstract model of the behavior of the BOM towards each action[2]. However external techniques are needed for the composability verification of BOM based components.

PN is an effective graphical tool and mathematical formalism for modeling concurrent systems and their behaviors. PN has existed for many decades and has been used in modeling, analysis and verification of a large variety of systems [5]. A PN is an algebraic structure of 3-tuple: PN = (P, T,φ) where:

- P is a finite set of places P = $\{p_1, p_2, \ldots, p_n\}$
- T is a finite set of transitions T = $\{t_1, t_2, \ldots, t_n\}$
- φ is a set of arcs $\varphi \subseteq (P \times T) \cup (T \times P) \mid P \cap T = \phi$ and $P \cup T \neq \phi$

A major strength of PN is its support for analysis of many properties and problems associated with concurrent systems. Some of the important PN properties are briefly defined as follows:

Reachability A marking μ is said to be reachable from a marking μ_0 if there exists a sequence of firings that transforms μ_0 to μ. The set of all possible markings reachable from μ_0 in a net P(N, μ_0) is denoted by R(N, μ_0). The reachability property determines whether $\mu_x \in R(N, \mu_0)$. Reachability is a fundamental basis for studying the dynamic behavior of a system [4].

Liveness A PN P(N, μ_0) is said to be live if, no matter what marking has been reached from μ_0 it is still possible to make further progress by firing an enabled transition of the net. A live PN guarantees deadlock-free operation, no matter what firing sequence is chosen [4]

In this paper we employ the well-suited analytical strength of PN tools to test various system properties such as deadlock freedom. We propose a verification process and a framework to allow the assessment of a composed model by transforming it into a PN model and apply suitable analytical methods which are commonly being practiced by the PN community. As compared to our previous approach[3] in which we used Finite State Automata, using PN for verification proves to be more fruitful due to the broader range of verification tools and techniques that are available. The remainder of this paper is organized as follows: Section 2 contains the major contribution of this work covering the proposed

methodology for the verification. A case study is presented in section 3 to support our concept whereas section 4 concludes this paper.

2 Test and Verification of a composed model

In this section we discuss our proposed verification process and the framework in which this process has been implemented. Our approach for the verification of BOM based composed models is essentially based on the conversion of a composed model to PN model and applying different analytical techniques to verify the required system properties. We use the Platform Independent PN Editor (PIPE) as an underlying layer for PN analysis [1]. PIPE is an open source Java based PN graphical editor and analysis tools library. It not only provides a graphical preview and editing facility for PN models, but also supports visual simulation (commonly known as Token game animation) and analysis techniques. Our verification framework is integrated with PIPE environment to utilize these facilities. Following are the two main steps in our proposed verification process:

2.1 Transformation of BOM to Petri

In this step, BOM is transformed into PN model. In the preparatory phase a composed BOM model is parsed using our BOM parser and the objects of state-machines, and corresponding events of all the members of composition are collected. Then the composition is analyzed to check completeness of model in terms of structure of state-machines and the presence of associated events for each state to be able to exit. In case of no such association it is verified whether the particular state is a final state or not. The event pairs are also matched among the member components so that for each receiver component a corresponding sender of an event is present. This procedure is applied to ensure that the composition is structurally correct and suitable to proceed with. For more details see [3].

Finally the BOM model is automatically transformed into a PN model. A PN model is ideally represented using PNML (PN Markup Language) which is an XML-based standard interchange format for PN. We propose the basic transformation algorithm as given in figure 2a. The input for this procedure is a BOM object C of the composed model which is generated by the parser. In order to meet the requirements of PNML, we also assume default values such that tokens at each place are set to zero and arc weights are set to 1. As we are not considering Timed PN in this work so we also set the timed Boolean variable to false. These settings can be modified later for experimentation at the time of execution & analysis.

In this algorithm for each state-machine in the composed model (line 2) every state is traversed one by one (line 3) and its corresponding place is created[1] in

[1] Create functions are used to write XML code to a file according to the PNML notation.

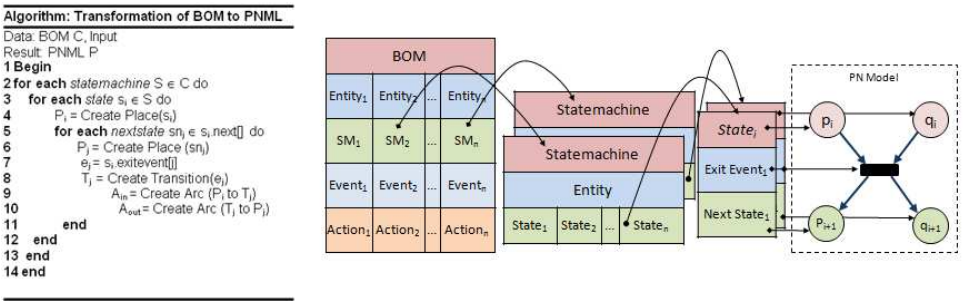


Fig. 1. a) Our Algorithm b) Transformation from BOM to PN

PNML format (line 4). Since a state may have multiple next states and their associated exit events so for each next state (line 5), a place is created (line 6) provided it does not already exist. Then a transition is created for the exit event associated with the next states (line 8). After that an input arc is created from the place to transition (line 9) and an output arc is created from the transition to the next place (line 10). Duplication is avoided in the entire process. This procedure can be graphically viewed in figure 2b, (assuming that the two state-machines have a common exit event) Also the graphical placing (X and Y positions) for each place, transition and arc is handled automatically in such a way that all the places of each member component are aligned in a vertical lane and their transitions are created between the lanes to show the interaction among different components in order to facilitate visibility. When the entire BOM model is traversed, a PNML file is generated that corresponds to the PNML standard and can be used in any PN simulation tool for execution.

2.2 Petri Net Analysis

In the analysis step, we perform Reachability analysis on the PNML model generated in the previous step with user given input parameters. To initiate this procedure we use PIPE library to construct a reachability tree and populate it with our PNML model generated in the previous step. We perform this operation programmatically to facilitate automation. After the initialization of the root marking, the tree recursively expands by constructing further markings until all the markings of the model have been created. If no more marking can be created, i.e., no further transition is enabled it reaches a dead marking. By detecting a dead marking in the tree, we assert that a deadlock is detected provided that it is not a final marking. We propose a minor change in the original definition of deadlock in a Dead (or L0-live) PN as some models may have terminal points e.g., when a Final State in a finite State-machine is reached. So we define a "Final Marking" as a successful termination point which is not considered as a dead marking even though no further transitions could be fired. Thus we propose a deadlock detection method in which we find at least one dead marking from

where no further transitions could be fired, provided it is not a final marking. We input the final marking as a parameter for reachability analysis.

3 Case Study

In this section we discuss a restaurant case study to test our verification approach. This is a simple composed model of two BOM components namely Customer and Waiter. In step 1, Restaurant BOM is parsed and the corresponding objects are generated and the model is transformed into PNML format using our proposed algorithm. Figure 2a represents the sequence diagram of the restaurant. Statemachines of customer and Waiter are presented in Figure 2b where as the generated PN is illustrated in 2c.

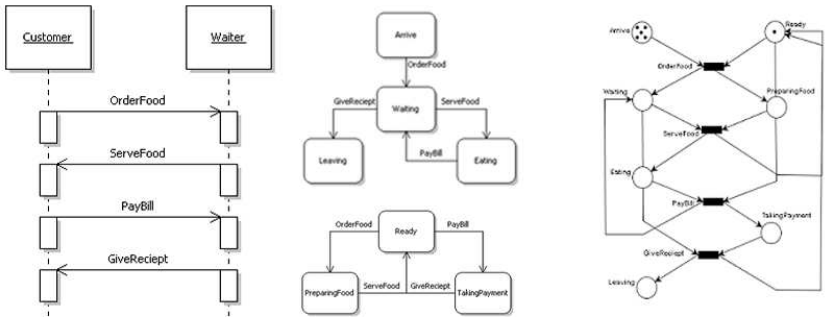


Fig. 2. a) Sequence Diagram b) Statemachine c)PN

When PN model is executed in PIPE simulator, a token is dispatched from the Arrive place representing the arrival of a customer who orders food. But it requires an available Waiter from Ready place to proceed. After Order Food transition is fired, one token will be produced to Waiting and the other will be produced to Preparing Food, place. In the same way, all the interactions between customer and waiter will continue, until the customers have reached Leaving whereas the waiter goes back to Ready. In step 2, the reachability analysis module is initiated and the final marking parameter [0,0,0,5,1,0,0] (i.e., all five customers leave the restaurant and the waiter is back to ready) is given as input to distinguish it from the dead markings. When the module completed the execution, it verified that the model is deadlock free as only one dead marking was detected in the tree which was exempted by the final marking parameter. The Reachability Tree can be viewed at: `http://web.it.kth.se/~imahmood/SimpleRestaurant.html`. In the tree each node (in blue) represents a marking and is denoted by a number and transitions are the arrows connecting next marking. From the tree it can be seen that node 50 (in green) is the only dead marking from where no further transition is possible. Thus we verify that although the model terminates it does not have a deadlock.

In a different experiment we introduced another component Kitchen in the composed model. Same procedure was applied however it was noted that few dead markings were detected because, there are some situations in which kitchen is in the cooking place and can be released only when waiter takes food to serve, whereas waiter is waiting for the kitchen to take more orders. This results in a potential deadlock which may or may not occur depending on the sequence of transitions fired. This potential deadlock is detected by our framework. The figures of the PN model and reachability tree of this experiment cannot be shown due to space limitations and can be viewed at: `http://web.it.kth.se/~imahmood/Restaurant.html` . We also encourage interested readers to view more case studies at our web site: `http://web.it.kth.se/~imahmood`

4 Summary and Conclusion

In this paper we discuss composability test of BOM based compositions using PN. We suggest an algorithm to transform a composed BOM model into a standard PN format known as PNML which can further be used with any PN analysis tool that conforms to this standard. We however suggest using PIPE tool in our framework and provide a technique to detect deadlocks in the composed model. Finally we explain the entire process using a Restaurant case study.

We are further interested to consider alternative techniques in the PN to extend the capability of our verification process, such as Timed PN and Colored PN to test and verify more realistic system models. We also are inclined to apply different state space reduction techniques to our framework in order to solve the state explosion problem and optimize the verification process.

References

1. Chung, E., Kimber, T., Kirby, B., Master, T., Worthington, M.: Platform independent petri net editor. Tech. rep., Imperial College, London, Project Report (2007)
2. Gustavson, P.: Guide for base object model (bom) use and implementation. Tech. Rep. SISO-STD-003-2006, Simulation Interoperability Standard Organizations (SISO), Orlando, FL USA (2006)
3. Mahmood, I., Ayani, R., Vlassov, V., Moradi, F.: Statemachine matching in bom based model composition. In: Proc. 13th IEEE/ACM Int. Symp. Distributed Simulation and Real Time Applications DS-RT '09. pp. 136–143 (2009)
4. Murata, T.: Petri nets: Properties, analysis and applications 77(4), 541–580 (1989)
5. Peterson, J.L.: Petri nets. Computing Surveys vol. Vol 9, no. No. 3 (September 1977)
6. Petty, M.D., Weisel, E.W.: A theory ofsimulation composability. Tech. rep., Virginia Modeling Analysis & Simulation Center, Old Dominion University, Norfolk, Virginia (2004)

Test Suite Reduction in Good Order: Comparing Heuristics from a New Viewpoint

Antonia Bertolino[1], Emanuela Cartaxo[2], Patrícia Machado[2], Eda Marchetti[1], and João Felipe Ouriques[2]

{antonia.bertolino,eda.marchetti}@isti.cnr.it
[1]ISTI-CNR, Via Moruzzi, 1, 56124. Pisa, Italy
{emanuela,patricia,jfelipe}@dsc.ufcg.edu.br,
[2]GMF-UFCG, Av. Aprígio Veloso, 882. Campina Grande, Brazil

Abstract. A new perspective in assessing test suite reduction techniques based on their rate of fault detection is introduced in this paper. This criterion, which is standard in assessing test-suite prioritization, has never been used for reduction. Our proposal stems from the consideration that under pressure testing could be stopped before all tests in the reduced test-suite are run, and in such cases the ordering in the reduced test-suite is also important. We compare four well-known reduction heuristics showing that by considering the rate of fault detection, the reduction technique to be chosen when time is an issue might be different from the one performing the best when testing can be completed.

Keywords: Heuristics, Test Ordering, Test Reduction

1 Introduction

Defining a suitable test suite that detects as many faults as possible is a challenge to software testers. To address it, test requirements are often defined as coverage criteria to be met, based on the assumption that a test suite that achieves the coverage is an effective one. Moreover, due to resource constraints, a test manager goal is to define a minimum such suite. In practice, test suites have a number of redundant test cases, particularly the ones that are automatically generated. In this sense, test suite reduction techniques have been investigated [1]. These techniques aim at reducing the size of a test suite as far as the test requirements can be reached. Therefore, test suite reduction is a key strategy to be applied independently of the kind of testing to be performed.

However, even this reduced test suite can be too big to meet resource constraints, and the tester may still face the problem of selecting only a few test cases. This issue has been mostly investigated in regression testing, where tests have to be periodically re-executed. In this case, prioritization techniques have been applied to define an order to execute the regression test cases by increasing the chances of early fault detection during retesting [2].

Several recent studies have compared reduction techniques considering their efficacy in decreasing test-suite size and their impact on fault detection effectiveness. Chen and Lau [3] present guidelines for choosing the most appropriate

Short Papers of the 22nd IFIP ICTSS, Alexandre Petrenko, Adenilso Simao, Jose Carlos Maldonado (eds.), Nov. 08-10, 2010, Natal, Brazil.

heuristic-based technique according to the size of the reduced suite. Zhong *et al.* [4] also consider genetic algorithms and the execution time of the suite. On the other hand, Rothermel *et al.* [5] focus on fault detection capability. Regarding model-based testing, Heimdahl and George [6] present a study based on coverage criteria such as Variable Domain, Transition, Decision, Decision Usage, MC/DC, MC/DC usage. Also considering coverage, Wong *et al.* [7] conclude that representative sets regarding fault detection capability can be reached. Finally, Harrold *et al.* [8] investigate separately two algorithms for test suite reduction and one for test suite prioritization taking as test requirement MC/DC coverage. Nevertheless, these studies analyze the reduced test-suite as a whole, without considering the order in which the tests are executed.

We consider here test ordering during test reduction, which is different from test prioritization during regression testing. The problem of executing first the most effective test cases while covering test requirements can be handled by following a test selection order based on the choices of the reduction technique, since they usually pick one-by-one a test case from the original test suite. In this sense, the best reduction technique to be applied in a particular context would be the one that meets the test requirements and also chooses the test cases in an order that favors the most relevant ones, particularly regarding fault detection effectiveness. To the best of our knowledge, how the effectiveness of a test reduction heuristic varies in relation to the number of test cases executed is something that has not be analyzed so far.

This paper proposes a new viewpoint in evaluating test suite reduction techniques by also considering the order by which test cases are selected (Section 2). For this, two real-world case studies are conducted by considering four well-known reduction heuristics (Section 3). To evaluate the results, we measure the rate of fault detection, which is commonly applied to assess prioritization techniques. As a result, some techniques may be more effective in selecting up to a certain number of test cases, even if they may not remain the best ones for a bigger number of test cases up to the limit of the complete reduced test suite (Section 4). Note that the goal is to illustrate that techniques can yield a different performance regarding fault detection when ordered subsets are considered and then to motivate evaluation of reduction techniques from this new viewpoint. Providing generally valid conclusions on which heuristic is the most effective is out of scope.

2 Background and Motivation

From Harrold *et al.* [1], considering generically a test requirement as a statement, a block, a decision, and so on, a test suite reduction can be defined:
Given: A test suite TS, a set of test requirements $Req = \{ Req_1, Req_2, \ldots, Req_n\}$ to be covered, and subsets of TS: $TS_1, TS_2, \ldots, TS_n$, where each test case of TS_i can be used to test Req_i: A representative subset RS of TS that satisfies all of the *Req's* must have at least one test case for each *Req*.

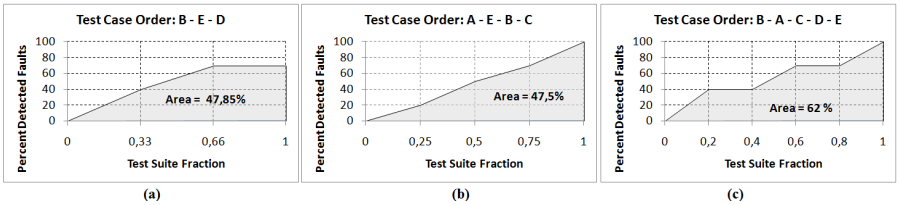


Fig. 1. Test Case Order

Thus the goal of test suite reduction is to construct a subset (RS) of the original test suite that still provides 100% coverage of test requirements.

Considering test case prioritization, the problem can be defined as follows [2]: **Given:** TS, a Test Suite; PTS, a set of permutations of TS; and, f, a function from PTS to real numbers. **Find:** $TS\prime \in PTS \mid \forall\ TS\prime\prime\ (TS\prime\prime \in PTS)\ (TS\prime\prime \neq TS\prime)\ ;\ f(TS\prime) \geq f(TS\prime\prime)$.

The objective function is defined from the goal of the prioritization. Then, a set of permutations PTS is obtained and the $PTS\prime$ that has the biggest $f(TS\prime)$ is chosen. When the goal is to increase fault detection, there is a metric largely used in the literature, named Average Percentage of Fault Detection – APFD. The highest APFD value, the fastest and the best the fault detection rates [2].

Usually heuristics for test suite reduction and prioritization are used in isolation. The ideal solution would be to maximize the number of failures detected while selecting a subset of non-redundant test cases that covers all requirements. To motivate such an approach, let us consider the same toy example presented in [2], where a program is supposed to contain 10 faults and a test suite of 5 test cases, called for simplicity (A,B,C,D,E), is available. When all test cases are executed, independently of their order, the percentage of fault detection is always 100%. To select the best prioritization technique, the APFD measure reached by the associated combination of test cases has been proposed. According to [2], the sets (A,B,C,D,E), (E,D,C,B,A), (C,E,B,A,D) yield respectively the following APFD values: 50%, 64 % and 84%. Hence the third one is the best.

Now, suppose we apply three different reduction techniques on the same test suite, obtaining the following reduced sets: TS1 = (B,E,D); TS2 = (A,E,B,C); TS3 = (B,A,C,D,E), all reaching 100% requirements coverage. TS1 performs the best in terms of test size reduction even if TS1 discovers the 70% of the faults, while TS2 and TS3 detect the 100%. As shown in Figure 1, the best technique considering the APFD measure would be TS3, but it is the one having the worst performance in terms of test suite reduction. A practical compromise should consider both the number of test cases executed and the rate of fault detection.

On the other hand, if for any reasons the test phase needs to be stopped after only two test case are executed, recalculating the APFD of the first two tests in TS1, TS2, TS3 the following results can be observed: TS1_r = (B,E) detects 7 faults and reaches the 35.5% of APFD; TS2_r = (A,E) detects 5 faults and reaches the 22.5% of APFD; TS3_r = (B,A) detects 4 faults and reaches the

30% of APFD. In this case the best APFD belongs to TS1 and not anymore to TS3. Thus if testing is stopped before all test cases in the reduced test-suite are run, the ordering in the reduced test-suite is important for selecting the most effective test strategy. This is why we propose to mix the concepts of reduction and prioritization.

3 Case Studies

In two real-world case studies, provided by Motorola, we compared four well-known test suite reduction heuristics – G, GE, GRE and H [3] – from the new viewpoint. The idea is to measure the rate of fault detection to show that the techniques based on these heuristics may present a different performance when considering the progressive coverage of selected test cases up to 100% coverage of the test requirements. For each application, Motorola Software Engineers elaborated the use case documents [9]. From these documents, Labelled Transitions Systems (LTS) are generated and from these, the test cases are obtained by using the LTS-BT tool [10]. We considered transition coverage as test requirement.

The applications selected for the case studies are: **Application 1** – TaRGeT, a desktop application – with 84 test cases that present redundancy (cover the same requirement); and **Application 2** – Direct License Acquisition, a feature for mobile phone applications – with 28 test cases that present redundancy, but each test case has at least one transition that is only covered by it.

The test cases were manually executed and the failures captured were associated with faults that can be detected by the suite. For **Application 1**, 13 faults have been defined, whereas for **Application 2**, 2 faults have been defined.

We have 4 reduced test suites (one for each heuristic) to be compared. Each heuristic was run 20 times. We collected the reduced test suite size for each execution and the number of the failures. For **Application 1**, the average of reduced test suite size for G, GE, GRE is 74 and for H is 74.45 and the number of faults are for G, GE, GRE and H, respectively, 10.4, 10.4, 10.5 and 10.75. For **Application 2**, the heuristics did not reduce the test suite, since each of the 28 test cases has at least 1 transition that is covered only by it. So, all heuristics kept on the reduced test suite the same number of test cases of the original test suite and, consequently, the rate of fault detection is not decreased.

To evaluate fault detection effectiveness, we constructed box plots to show the distribution of faults in 20 executions. For lack of space, we only report the box plots obtained for **Application 1** respectively for heuristics GE, GRE, Greedy and H (Figure 2-5).

For **Application 1**: if the tester is able to execute only 1-5 test cases, then GE and GRE is the best choice (1 fault detected when compared to 0-1 for the others). From 6-20 test cases, G is the best choice (from 3 to 7 faults when compared to from 2 to 6 for the others), whereas for more than 20 test cases, H is the best choice. For **Application 2**: if the tester is able to execute only 1-10 test cases, then H is the best choice (1-2 faults detected when compared to 0-1 for the others). For more than 10 test cases, GE and GRE are the best

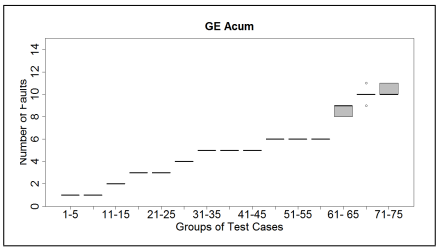


Fig. 2. Application 1 - GE

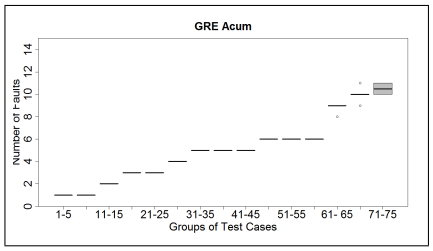


Fig. 3. Application 1 - GRE

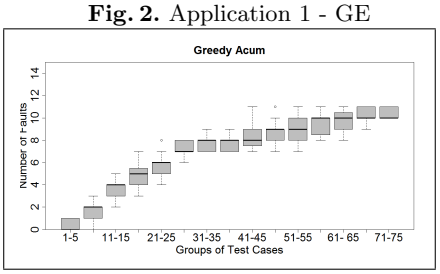


Fig. 4. Application 1 - Greedy

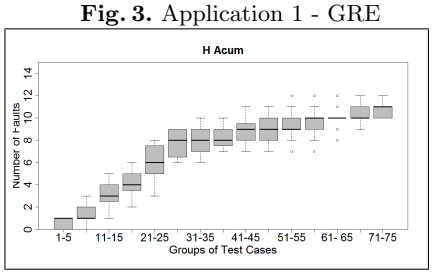


Fig. 5. Application 1 - H

choice (always detect 2 faults). It is important to highlight that GE and GRE heuristics present the same behaviour for all positions.

From these case studies, it can be noticed that by analysing the rate of fault detection, it is possible to observe which technique can be more effective, depending on the goals of the tester. Some techniques are more effective when only the first selected test cases can be handled, whereas others improve their performance as more test cases are considered.

Of course, the results observed cannot be generalised to other applications different from the two case studies considered here. The presented case studies are just meant to illustrate the differences that may arise. Wider experiments need to be executed to get more general conclusions.

4 Conclusions

Common practice to compare reduction techniques considers that all the test cases in the reduced test suite will be executed. However, if under budget constraints testing is stopped in advance than planned, the test methodology chosen for selecting the test cases during the planning could not be anymore the best choice. In this paper we showed how the fault detection effectiveness of the reduction heuristics could change when testers are forced to drastically reduce the number of test cases scheduled for a certain software. We considered four well-known reduction heuristics – G, GE, GRE and H – and measured their progressive rate of fault detection on two real-world applications.

The analysis of the case studies evidences how the performance of the four heuristics can be really influenced by the number of the test cases executed.

Of course the studies performed so far cannot be used for general conclusions and further investigations are necessary. Our goal was to show that the metrics adopted so far for assessing the relative effectiveness of various reduction approaches probably do not completely match a reality in which the testing phase can be shortened depending on time and cost constraints. Our results suggest that probably a new way of measuring the performance of various heuristics could be necessary. Identifying such a new assessment approach is part of our future work as well as executing more case studies and experiments.

5 Acknowledgements

This work was partially supported by the National Institute of Science and Technology for Software Engineering (INES[1]), funded by CNPq, grant 573964/2008-4 and CT-INFO/CNPq (Process 140074/2008-2).

References

[1] Harrold, M.J., Gupta, R., Soffa, M.L.: A methodology for controlling the size of a test suite. ACM Trans. Softw. Eng. Methodol. **2** (1993) 270–285

[2] Elbaum, S., Malishevsky, A.G., Rothermel, G.: Prioritizing test cases for regression testing. In: In Proc. of the Int. Symposium on Soft. Test. and Analysis, ACM Press (2000) 102–112

[3] Chen, T.Y., Lau, M.F.: A simulation study on some heuristics for test suite reduction. Information & Software Technology **40** (1998) 777–787

[4] Zhong, H., Zhang, L., Mei, H.: An experimental comparison of four test suite reduction techniques. In: ICSE '06: Proc. of the 28th Int. Conf. on Soft. Engineering, New York, NY, USA, ACM (2006) 636–640

[5] Rothermel, G., Harrold, M.J., Ronne, J.V., Hong, C.: Empirical studies of test-suite reduction. Journal of Soft. Test., Verification, and Reliability **12** (2002) 219–249

[6] Heimdahl, M.P.E., George, D.: Test-suite reduction for model based tests: Effects on test quality and implications for testing. In: ASE '04: Proc. of the 19th IEEE Int. Conf. on Automated Soft. Engineering, Washington, DC, USA, IEEE Computer Society (2004) 176–185

[7] Wong, W.E., Horgan, J.R., Mathur, A.P., Pasquini, A.: Test set size minimization and fault detection effectiveness: A case study in a space application. Journal of Systems and Software **48** (1999) 79–89

[8] Jones, J.A., Harrold, M.J.: Test-suite reduction and prioritization for modified condition/decision coverage. IEEE Trans. Softw. Eng. **29** (2003) 195–209

[9] Nogueira, S., Cartaxo, E., Torres, D., Aranha, E., Marques, R.: Model based test generation: An industrial experience. In: 1st Workshop on Systematic and Automated Soft. Testing - SBBD/SBES 2007, Joao Pessoa, PB, Brazil (2007)

[10] Cartaxo, E.G., Andrade, W.L., Neto, F.G.O., Machado, P.D.L.: LTS-BT: a tool to generate and select functional test cases for embedded systems. In: SAC '08: Proc. of the 2008 ACM Symposium on Applied Computing. Volume 2., New York, NY, USA, ACM (2008) 1540–1544

[1] www.ines.org.br

A Multiobjective Tabu Search Algorithm for Reducing Mutation Test Costs

Adam Banzi, Gabriel Pinheiro, João Carlos Árias, Tiago Nobre, Aurora Pozo, and Silvia Regina Vergilio *

Computer Science Department, Federal University of Paraná
CP: 19081, CEP 19031-970, Curitiba, Brazil
{adam,gabrielb,joao,tiagon,aurora,silvia}@inf.ufpr.br

Abstract. Some strategies were introduced in the literature to decrease mutation test costs, and to determine sets of essential mutation operators to reduce the number of mutants to be executed without decreasing the mutation score. However, the operator selection, in practice, may include real constraints and is dependent on diverse factors. In fact this is a multiobjective problem, that is, this problem does not have a single solution. Different sets of operators exist for multiple objectives to be satisfied, and some restrictions can be used to choose among the existing sets. To make this choice possible, we introduce in this paper a multiobjective approach based on a Tabu search algorithm. The algorithm was evaluated in a set of C programs and generated a set of solutions that dominate the solutions obtained by other traditional strategies.

Keywords: fault-based test, essential mutation operators

1 Introduction

The mutation score is an important measure to evaluate the quality of the test cases. It is usually obtained by executing a lot of mutant programs generated by a set of operators that describe faults. However, for a program, some operators can generate unnecessary and redundant mutants, associated to test cases that do not contribute to increase the score, and are not supposed to reveal additional faults. This makes mutation test very expensive, and to reduce its costs, some strategies were proposed. For example: Randomly Selected X% Mutation [1]; Constrained Mutation [9]; and N Selective Mutation. Other works investigate the determination of essential operators for Fortran [7] and C languages [2, 8] to reduce the number of mutants to be executed without decreasing the global score.

However, the mentioned works that address the selection of essential operators present some limitations. The first one [6] is that the experiments described by most works were conducted by using small programs with simple data and control structures. Another problem is that the strategies to determine the set

* This work is partially supported by CNPq

of operators are empirical, and provide at the end only a specific set of essential operators, to maximize the mutation score with a low number of mutants. They are not able to generate different good sets according to different objectives. The operator selection, in practice, may include real constraints and is dependent on diverse factors besides the number of mutants and score, such as: number of test data, execution time, number of revealed faults, number of equivalent mutants, and so on. In fact this is a multiobjective problem, that is, this problem does not have a single solution. Different sets of operators exist for multiple objectives to be satisfied, and some restrictions can be used to choose among the existing sets. According to the test objectives and resources, we can desire to use a different essential set of operators that prioritize a certain factor.

To make this choice possible, we introduce in this paper a multiobjective Tabu-search algorithm. To allow comparison with the traditional strategies, we illustrate the use of the algorithm in the unit test of real C programs, with two objectives: number of mutants and score. Graphical analysis shows that the solutions presented by the algorithms are better than the solutions of the traditional strategies considering both objectives.

This paper is organized as follows. Section 2 introduces the multiobjective formulation of the mutant operator selection problem and describes the implemented algorithm. Section 3 presents experimental results. Section 4 concludes the paper and presents new directions of research.

2 The Multiojective Approach

Optimization problems with two or more objective functions are called multiobjective. In such problems, the objectives to be optimized are usually in conflict, which means that they do not have a single solution. In this way, the goal is to find a good trade-off of solutions that better represent the possible compromise among the objectives, that is, solutions that are not dominated by any other in the objective space. A set of non-dominated objective solutions is called Pareto optimal and the set of all non-dominated solutions is called Pareto Front.

We can observe that the problem of selecting a set of operators is in fact multiobjective. It may involve a number Q of different objectives to be optimized: mutation score, number of test cases, number of mutants, number of equivalent mutants, number of faults to be revealed, execution time, and so on. Given a program P, a set of mutation operators O, and a vector of Q objective functions, $f_i, i = 1...q$. The problem is to find a subset O', such that O' is the Pareto optimal set with respect to the objective functions, $f_i, i = 1...q$.

The chosen representation for the problem is simple, with the solution being represented by a vector whose positions assume an integer number in the interval $[1, N]$, being N the number of the mutation operators. Each operator can not appear more than once in the vector.

However, the front is dependent on the program, but, it is very expensive to execute the optimization algorithms every time that the test occurs, because of this, the essential sets of operator can be previously established from the

analysis of different fronts obtained for similar programs (in the same application domain). To allow comparison with traditional strategies, in this work, we will take a benchmark, composed by six C programs from the same domain and will consider unit test and two objectives: number of mutants and mutation score. Hence, the problem is the search for sets of mutation operators that maximize the mutation score and minimize the number of mutants.

The Tabu Search approach was proposed by Glover [4] to allow hill climbing and avoid local optima. The basic principle of Tabu Search is to pursue the search whenever a local optimum is encountered by allowing non-improving states. Cycling back to previously visited solutions is prevented by the use of memories, called tabu lists, that keep the recent history of the search. In [3] a Multiobjective version of the Tabu algorithm was proposed, the MTabu algorithm (Multiobjective optimization using Tabu search), which runs over a neighborhood using Pareto's concepts. The algorithm works with the following lists: *candidateList* - List that receives all the non dominated neighbors of the current solution; *paretoList* - List that contains already explored non dominated solutions; and *tabuList* - List that keeps all explored solutions.

The MTabu algorithm (see Algorithm 1) begins generating a random solution (`SeedSolution`). This solution, called *currentSolution*, is expanded and the neighborhood is kept in *candidateList*. After then, *paretoList* and *tabuList* are updated with the *currentSolution*. These steps are an iteration of the inner loop. This inner loop is repeated a number of times depending on the maximum number of iterations (*max_iterations*). An external loop controls the number of different randomly generated solutions (*maxseed*) to be explored.

Algorithm 1 Pseudocode of MTabu.

```
Initialize numberseed, candidateList, paretoList, tabuList;
while numberseed < maxseed do
  counter = 0; currentSolution = SeedSolution(numberseed);
  while counter < max_iterations do
    for i = 1 to neighborhoodSize do
      candidate = neighbor(currentSolution);
      if (not contains(tabuList, candidate)) then
        Evaluate(candidate);
        Update(candidateList, candidate);
      end if
    end for
    Update(tabuList, currentSolution);
    Update(paretoList, currentSolution);
    currentSolution = randomlySelectedCandidate(candidateList);
    counter =+ ;
  end while
  numberseed =+ ;
end while
```

3 Experiments

We used a set of C programs from SIR [5]. These programs have been used as benchmark for many software testing works [6]. All the test data available in the repository were submitted to Proteum, version 1.4 (Linux), and the 71 operators of this version were used. The alive mutants were considered equivalent. The test cases that did not contribute to increase the mutation score in the order of execution were also discarded. At the end, we obtained for each program a test set T that we will call M-adequate.

We applied some well known strategies to reduce the number of mutants. Each strategy applies a subset of operators, $O' \subseteq O$, which generates a number M' of mutants. After this, we determined a set $T' \subseteq T$ that is M'-adequate, that is, test cases of T that really contribute to increase the mutation score in the same initial execution order. The score obtained considering M and T' was then calculated and used in the comparison.

In the sequence, we provide a brief description of how the sets O' and M' for each strategy were obtained: a) Randomly Selected X% Mutation: this strategy was implemented to randomly select $10\%, 20\%$ and 40% of the mutants generated by each mutation operator. Hence, $O' = O$; b) N Selective Mutation: we considered three values for N: $5, 10$ and 20. Then, the mutants generated by $N = 5$ operators that generated more mutants were excluded from the total set M, and a set O' was used, with $(O - 5)$ operators. Similarly for the other values of N; c) Essential operators for unit test from Barbosa et al [2]: using the following set of operators: {SWDD, SMTC, SSDL, OLBN, ORRN, VTWD, VDTR, CCCR, CCSR}. In this case $O' = 9$.

The Tabu-search algorithm was run ten times with: $maxseed$=5; $max\_iterations$= 500; $tabuListSize$=20; $neighborhoodSize$=10. The Tabu fronts are displayed in Figures 1 to 6, and are used in the comparison with the traditional strategies. To a better visualization, some strategies with bad results do not appear in the figures. We represent only the best points from the applied strategies.

For `printtokens`, the best coverage result of the traditional strategies was obtained by the Selective ($N = 5$) (score equal to 1), but, the number of mutants is very high (1774), because of this, the correspondent point is not displayed in the graph. MTabu reaches the same coverage with a lower number of mutants (525). This result is also better, in terms of coverage and number of mutants, than the results of Random ($N = 20\%$), Random ($N = 40\%$), Selective ($N = 10$) and Essential operators. The strategies Random ($N = 10\%$) and Selective ($N = 20$) have results with lower number of mutants but lower coverage too, in this case, the solution of MTabu (with score = 0.9987 and number of mutants = 271) dominates both.

For `printtokens2`, the best coverage result of the traditional strategies, was obtained by using the Essential operators (score equal to 1), but again, the number of mutants is very high (1221). MTabu reaches the same coverage with a lower number of mutants (535). This result is better than that ones of Random ($N = 20\%$), Random ($N = 40\%$), Selective ($N = 5$) and Selective ($N = 10$). The strategies Random ($N = 10\%$) and Selective ($N = 20$) have results with lower

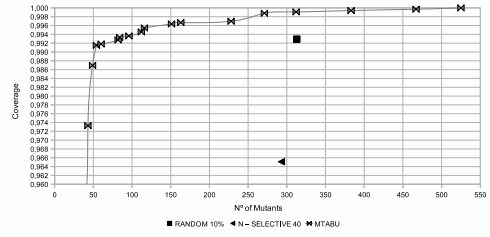


Fig. 1. Program `printtokens`.

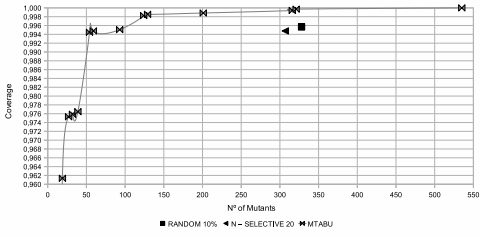


Fig. 2. Program `printtokens2`.

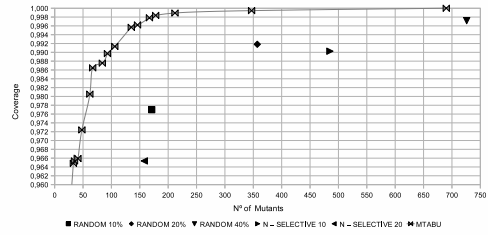


Fig. 3. Program `schedule`.

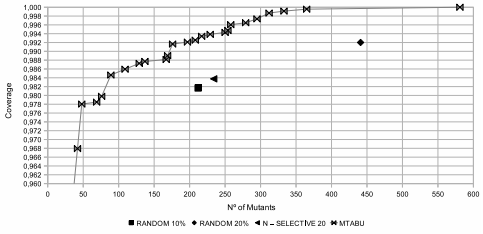


Fig. 4. Program `schedule2`.

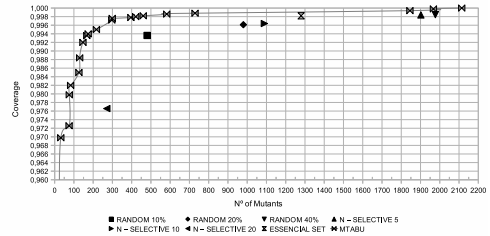


Fig. 5. Program `totinfo`.

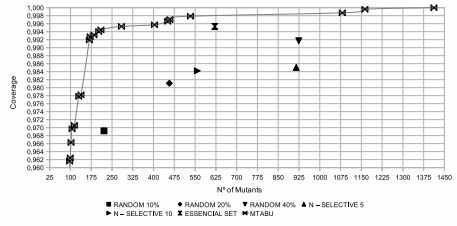


Fig. 6. Program `tcas`.

number of mutants but lower coverage too, in this case, the solution of MTabu (with score = 0.9988 and number of mutants = 195) dominates both.

The remarks made above are valid to the other programs. For `schedule`, the best results of the traditional strategies, in terms of mutation score, was obtained by Selective ($N = 5$) (score equal to 0.9989) with a high number of mutants (907). MTabu reaches the score equal to 1 with a lower number of mutants (690). This result is better than that one obtained by Random ($N = 40\%$). The other strategies have lower coverage and always it is possible to obtain better results with MTabu as observed in Figure 3. In sum, from all figures we observe that the results provided by the traditional strategies are always dominated by the results of MTabu.

4 Conclusions

This paper described a multiobjective algorithm using Tabu search (MTabu) to the mutation operator selection problem. To allow comparison with the traditional approach, two objectives were used: number of mutants and mutation score. The results provided by the traditional strategies are always dominated by the results of MTabu, or in other words, there are a lot of solutions that are better than that ones obtained by the traditional strategies. Differently of the traditional strategies, the approach allows that other factors that may influence the selection can be considered. Each solution represents a set of operators, and by analysing the fronts, different sets of essential operators can be generated. A set should be chosen according to the real test restrictions. If the tester is interested in a balance among the objectives, all the solutions can be used. However, if an objective has higher priority, the best solutions considering such objective should be taken.

A procedure to determine a set of essential operators according to priorities given to each objective is now being automated, and other evaluation experiments should be conducted. Now, we are investigating the use of three objectives considering the minimization of equivalent mutants. Other research directions include to evaluate objective functions related to the characteristics of the programs being tested, and to explore the use of the approach in other contexts, such as: mutation interface and Java programs.

References

1. Acree, A., Budd, T., DeMillo, R., Lipton, R., Sayward, F.: Mutation analysis. School of Information and Computer Science, Georgia Institute of Technology (1979)
2. Barbosa, E., Maldonado, J., Vincenzi, A.: Towards the determination of sufficient mutant operators for C. Sw Test. Verification and Reliability 11, 113–136 (2001)
3. Baykasoglu, A., Owen, S., Gindy, N.: A Taboo Search Based Approach to Find the Pareto Optimal Set in Multiple Objective Optimisation. Journal of Engineering Optimization pp. 731–748 (1999)
4. Glover, F.: Future Paths for Integer Programming and Links to Artificial Intelligence. Comput. Open Res. pp. 13:533–549 (1986)
5. Hutchins, M., Foster, H., Goradia, T., Ostrand, T.: Experiments of the effectiveness of data data flow and control flow-based W-test adequacy criteria. In: 16th International Conference on Software Engineering (ICSE 1994). pp. 191–200 (1994)
6. Namin, A.S., Andrews, J.H., Murdoch, D.: Sufficient Mutation Operators for Measuring Test Effectiveness . In: ICSE'2008. pp. 351–360 (2008)
7. Offutt, A., Lee, A., Rothermel, G., Untch, R., Zapf, C.: An experimental determination of sufficient mutant operators. ACM Transactions on Software Engineering and Methodology 2(5), 99–118 (1996)
8. Vincenzi, A., Maldonado, J., Barbosa, E., Delamaro, M.: Essential Interface Operators: A Study Case. In: Proceedings of the 13th Brazilian Symp. on Software Engineering. pp. 373–391 (1999), in Portuguese
9. Wong, W., Mathur, A.: Reducing the cost of mutation testing: An empirical study. The Journal of Systems & Software 31(3), 185–196 (1995)

Automatic Test Generation for Data-Flow Reactive Systems with time constraints*

Omer Nguena Timo[1], Hervé Marchand[2], and Antoine Rollet[1]

[1] LaBRI (University of Bordeaux - CNRS), Talence, France
[2] INRIA, Centre Rennes-Bretagne Atlantique

Abstract. In this paper, we handle the problem of conformance testing for data-flow critical systems with time constraints. We present a formal model (Variable Driven Timed Automata) adapted for such systems inspired from timed automata using variables as inputs and outputs, and clocks. In this model, we consider urgency and the possibility to fire several transitions instantaneously. We present a conformance relation for this model and we propose a test generation method using a test purpose approach, based on a region graph transformation of the specification.

1 Conformance testing with VDTA

We define the conformance testing relation **tvco** for Data-flow reactive systems. such systems are characterized by the fact that they interact with their environment in a continuous way by means of continuous input and output set of events (taking their values in (possibly) infinite domains), while obeying some timing constraints. In this framework, continuous means that the values of the inputs events can be updated at anytime, while the value of the outputs events can always be observed. We choose to model our sytems by Variable Driven Timed Automata (VDTA) [7], a variant of timed automata [2] with only urgent transitions (i.e fired as soon as constraints are true). VDTA rather uses variables communication mechanism than synchronising actions. We first introduce the VDTA model and then our conformance relation **tvco**.

1.1 Variable driven timed automata (VDTA)

Given a set of variables V, each variable $V_i \in V$ ranges over a (infinite) domain $Dom(V_i)$ in $\mathbb{N}$, $\mathbb{Q}_+$ or $\mathbb{R}_+$. We denote $\mathcal{G}(V)$ the set of variable constraints defined as boolean combinations of constraints of the form $V_i \bowtie c$ with $V_i \in V$, $c \in Dom(V_i)$ and $\bowtie \in \{<, \leq, =, \geq, >\}$. For a set V, a variable assignment for V is a tuple $\Pi_{i\in[1..n]}(\{V_i\} \times (Dom(V_i) \cup \{\perp\}))$ and we denote by $A(V)$ the set of variable assignments for V. Given $G \in \mathcal{G}(V)$ and a valuation $v \in Dom(V)$, we write $v \models G$ when $G(v) \equiv true$. Given a valuation $v = (v_1, \cdots, v_n)$ of V and $A \in A(V)$, we define the valuations $v[A]$ as $v[A](V_i) = c$ if $(V_i, c) \in A$ and $c \neq \perp$, and $v[A](V_i) = v_i$ otherwise. Intuitively, (V_i, c) of A requires to assign c to V_i if c is a constant from $Dom(V_i)$; otherwise c is equal to $\perp$ and *no access* to V_i should be done. The identity $Id_V \in A(V)$ lets unchanged all the variables of V. We define the constraint $G[A] = \{v[A] \mid v \models G\}$. $Proj_{V_i}(G)$ denotes the constraint such that $(v_1, \cdots, v_{i-1}, v_{i+1}, \cdots, v_n) \models Proj_{V_i}(G)$ iff $\exists v_i \in Dom(V_i)$ such that $(v_1, \cdots, v_n) \models G$. We extend it to a subset V' of V and we denote it $Proj_{V'}(G)$.

* This work was supported by French National Agency ANR Testec Project.

Definition 1 (VDTA). *A* Variable Driven Timed Automaton *(VDTA) is a tuple* $\mathcal{A} = \langle L, X, I, O, l^0, G^0, \Delta_\mathcal{A} \rangle$*, where* L *is a finite set of* locations*,* $l^0 \in L$ *is the* initial location*,* $G^0 \in \mathcal{G}(I, O)$ *is the* initial condition*, a constraint with variables in* $I \cup O$ *and transitions in the transition relation* $\Delta_\mathcal{A} \subseteq L \times \mathcal{G}(I, O, X) \times A(O) \times 2^X \times L$ *:* $\langle l, G, A, \mathcal{X}, l' \rangle \in \Delta_\mathcal{A}$ *is a* transition such *that*

- $G \in \mathcal{G}(I, O, X)$ *is a boolean combination of elements of* $\mathcal{G}(I)$*,* $\mathcal{G}(O)$ *and* $\mathcal{G}(X)$ *whereas* $A \in A(O)$ *is an assignement on output variables.*
- $\mathcal{X} \in 2^X$ *is a set of clocks that are reset when triggering the transition.*

In the sequel, we write $l \xrightarrow{G,A,\mathcal{X}} l'$ when $\langle l, G, A, \mathcal{X}, l' \rangle \in \Delta_\mathcal{A}$. $\mathcal{A}$ is *deterministic* if G_0 is satisfied by at most one valuation (i^0, o^0); and for all $l \in L$, for all G_1, G_2 such that $l \xrightarrow{G_i,A_i,\mathcal{X}_i} l_i$ s.t. $G_1 \neq G_2$, $G_1 \cap G_2$ is unsatisfiable. A *state* of a VDTA as above defined is of the form (ℓ, i, o, x) where i, o and x are *valuations* of input, output and clock variables. A valuation $v \in Dom(V)$ is simply a function that returns the values of the variables in V. If $A \in A(I)$ is an assignement on input variables, the valuation $i[A]$ change the value of input variables according to the assignement. If x is clock valuation, $\mathcal{X}$ is a subset of clocks, and $\delta \in \mathbb{R}_+$ a delay, the valuation $x + \delta$ add δ to each clock value and the valuation $x[\mathcal{X}' \leftarrow 0]$ resets from x all clocks in $\mathcal{X}$. These notions are standard in the literature.

Definition 2. *The semantics of a VDTA* $\mathcal{A} = \langle L, X, I, O, l^0, G^0, \Delta_\mathcal{A} \rangle$*, is a TTS defined by the tuple* $[\![\mathcal{A}]\!] = \langle S, s^0, \Sigma, \rightarrow \rangle$ *where* $S = L \times Dom(I) \times Dom(O) \times \mathbb{R}_+^X$ *is the (infinite) set of states,* $s^0 = (l^0, i^0, o^0, x^0)$ *is the initial configuration where* x^0 *is the clock valuation that maps every clock to* 0 *and* (i^0, o^0) *is the only solution of* G^0*,* $\Sigma = A(I) \cup A(O) \cup \mathbb{R}_+^X$ *is the (infinite) set of actions, and* $\rightarrow$ *is the transition relation with the following three types of transitions:*

T1 $(l, i, o, x) \xrightarrow{A} (l', i, o[A], x[\mathcal{X} \leftarrow 0])$ *if there exists* $(l, G, A, \mathcal{X}, l') \in \Delta_\mathcal{A}$ *such that* $(i, o, x) \models G$*,*

T2 $(l, i, o, x) \xrightarrow{A} (l, i[A], o, x)$ *with* $A \in A(I)$ *if* $\forall (l, G, A', \mathcal{X}, l') \in \Delta_\mathcal{A}$*,* $(i, o, x) \not\models G$*.*

T3 $(l, i, o, x) \xrightarrow{\delta} (l, i, o, x + \delta)$ *with* $\delta > 0$ *if for every* $\delta' < \delta$*, for every symbolic transition* $(l, G, \mathcal{X}', l') \in \Delta_\mathcal{A}$*, we have* $(i, o, x + \delta') \not\models G$*.*

The environment (e.g. the tester) of a system modeled by a VDTA observes all the variables; it can assign a value to an input in I by performing a transition of type **T2**. Only the system can assign values to outputs in O by performing a transition of type **T1**. Transitions in $\mathcal{A}$ are urgent and *output-update transitions* **T1** are taken prior to *input-update* **T2** and *time-elapsing* **T3** transitions.

Notations. We denote $\rightarrow_{T_i}$ for transitions of type $T_i, i = 1...3$. $\text{Out}(s) = o$ gives access to the output value of $[\![\mathcal{A}]\!]$ in state $s = (l, i, o, x) \in S$. We write $s \xrightarrow{A}$ when there exists s' such that $s \xrightarrow{A} s'$ otherwise we write $s \not\xrightarrow{A}$. The latter notation is standard and it extends to sequences in Σ^*. A *run* is a sequence of alternating states and actions $s = s_0 a_1 s_1 \cdots a_n s_n$ in $S.(\Sigma.S)^*$ such that $\forall i \geq 0, s_i \xrightarrow{a_{i+1}} s_{i+1}$. $Run(s, [\![\mathcal{A}]\!])$ denotes the set of runs that can be executed in $[\![\mathcal{A}]\!]$ starting in state s and we let $Run([\![\mathcal{A}]\!]) = Run(s^0, [\![\mathcal{A}]\!])$. The *trace* $\rho(r)$ of a run $r = s_0 a_1 s_1 \cdots a_n s_n$ is given by the sequence $\rho(r) = Proj_S(r) = a_1 \cdots a_n \in \Sigma^*$.

$Tr([\![\mathcal{A}]\!]) = \{\rho(r) | r \in Run([\![\mathcal{A}]\!])\}$ is the set of traces generated by $\mathcal{A}$. We note $CoReach(S)$ the set of states from which S can be reached in $[\![\mathcal{A}]\!]$.

Stable VDTA. Thanks to their priority, several output-update transitions can be triggered in null delay. A *stable state* is a state from which no output-update transition can be fired. Formally a state s of $[\![\mathcal{A}]\!]$ is *stable* whenever for every $A \in A(O)$, $s \overset{A}{\not\longrightarrow}$. To leave this state, either the input values need to be updated or we need to let the time elapse. A *stable run* is a run that ends in a stable state. A VDTA $\mathcal{A}$ is *stable* if there is no loop of unstable states in $[\![\mathcal{A}]\!]$. In the sequel, we shall only consider stable VDTA.

Relation with other models. Timed automata [2] and most of its extensions (with delayable, eager [3], lazy transitions and variables [1] or UPPAAL model) use synchronising actions even for passing variable communication values. But the use of synchronising actions is not adequate for variable-based communication systems as the models should describe all synchronisations with the environment and they become rather unclear and big. To our knowledge urgency and variable passing mechanism are not accurately studied in timed systems for model-based testing. Results in [3] show that timed automata and urgent timed automata (with only clock variables) are equivalent in a language point of view.

1.2 Conformance testing relation

Conformance testing consists in checking that an implementation exhibits an observable behavior consistent with its specification. The main idea of conformance relation is that all observable behaviours of the implementation have to be allowed by the specification. Especially:

1. An implementation is not allowed to update a variable in a time (too late or too early) when it is not allowed by the specification.
2. An implementation is not allowed to omit to change a memory-variable at the time it is required by the specification.

A general conformance testing activity consists in executing sequences of input and delays and in checking whether the output of the implementation coincide with those of the specification. The **tioco** relation [5] (a timed extension of **ioco** [9]) requires to observe all synchronising output actions on sequences (even of duration equals to zero) whereas the conformance relation for VDTA considers the values of outputs in states where many assignements on outputs (state changing) can occur in zero time unit. Each of these output assignement can or cannot be observed depending on the granularities (frequency) of clocks of the tester and the implementation. We assume the same granularity and thus the outputs are observed only when the implementation is in *stable* states. Given a stable state s, we will thus be interested in the next stable states (or a singleton if the VDTA is deterministic) the implementation can reach from s after the execution of an input-update $A_i \in A(I)$. Given two stable states $s, s' \in S$, we write:

- $s \overset{A_i}{\Longrightarrow} s'$ if there exists a sequence $\sigma \in A(O)^*$ s.t $s \overset{A_i}{\longrightarrow} s'' \overset{\sigma}{\rightarrow} s'$, i.e. s' is the unique stable state that can be reached from s after updating the input variables with A_i, only triggering urgent transitions in zero time unit.

– $s \stackrel{\delta}{\Longrightarrow} s'$ if there is a sequence $\sigma = \sigma_1 \cdots \sigma_n \in (\{Id_O\} \cup \mathbb{R}_+)^*$ s.t $s \stackrel{\sigma}{\rightarrow} s'$ and $\delta = \sum_{\delta_i \in Proj_O(\sigma)} \delta_i$, i.e. s' is the stable state that can be reached by letting the time elapse during δ units of time with no observable output update.

The timed transtion system $Obs(\mathcal{A}) = (S, s^0, A(I) \cup \mathbb{R}_+, \Longrightarrow)$ is inductively generated from $[\![\mathcal{A}]\!]$ by starting from s^0 (that is supposed to be stable) and by using the two previous rules. We let $ObsRun(\mathcal{A}) = Run(Obs(\mathcal{A}))$ and $ObsTr(\mathcal{A}) = Tr(Obs(\mathcal{A}))$ denote the set of observed runs and observed traces of $\mathcal{A}$. Finally, we define $s\ \mathit{Safter}\ \alpha = \{s' \mid s \stackrel{\alpha}{\Longrightarrow} s'\}$ and $\mathcal{A}\ \mathit{Safter}\ \alpha = s^0\ \mathit{Safter}\ \alpha$.

We assume that the implementation Imp and the specification $\mathcal{A}$ are both modeled by compatible VDTA (i.e. they share the same input and output variables).

Definition 3. *Imp conforms to $\mathcal{A}$ (Imp tvco $\mathcal{A}$) whenever*

$$\forall \sigma \in ObsTr(\mathcal{A}), Out(Imp\ \mathit{Safter}\ \sigma) \subseteq Out(\mathcal{A}\ \mathit{Safter}\ \sigma)$$

This relation intends to check if the values of output variables of the implementation are correct after any sequence made of assignments of input variables or time elapsing. The tester can observe all output variables of the implementation. The tester can only update the input variables of the implementation or let the time elapse. Note that the blocking aspect as in [9] is not considered so far.

2 Reachability analysis

When testing, it is sometimes useful to target some particular states of the systems. To do so, we shall use a backward reachability analysis algorithm on the so-called *timed abstract graph* based on the known *region* abstraction [2]. We let $Reg(X)$ be the finite set of clock regions with respect to K, the maximal constant clocks in $\mathcal{A}$ are compared with. $[x]$ denotes the clock region that contains x, and $[\![r]\!]$ denotes the set of clock valuations whose clock region is equal to r. $I\_Succ(r)$ denotes *the immediate successor* of r. A region can be represented by a *diagonal clock constraint* that involves comparisons of two clocks. If r is a region, then G_r denotes the unique atomic (smallest) clock constraint such that $r \subseteq [\![G_r]\!]$. Given a location ℓ and a region r, $Gds_{\mathcal{A}}(\ell, r) = \{G \mid \ell \xrightarrow{G,A,\mathcal{X}} \ell' \text{ and } [\![r]\!] \subseteq [\![G]\!]\}$ is the set of constraints whose timing part is satisfied by r.

Definition 4 (Time-abstract graph). *The* time-abstract graph (TAG) *of a VDTA $\mathcal{A}$ as above, is the VDTA $RG(\mathcal{A}) = \langle Reg(\mathcal{A}), X, I, O, (\ell^0, r^0), G^0, \Delta_{RG}\rangle$ where $Reg(\mathcal{A}) = L \times Reg(X)$ is the set of locations of $RG(\mathcal{A})$, The transition relation, $\Delta_{RG} \subseteq Reg(\mathcal{A}) \times \mathcal{G}(I, O, X) \times A(O) \times Reg(\mathcal{A})$ is such that:*

U1 $(\ell, r) \xrightarrow{Proj_X(G) \wedge G_r, A, \mathcal{X}} (\ell', r')$ *iff* $\exists \ell \xrightarrow{G,A,\mathcal{X}} \ell'$ *s.t.* $[\![r]\!] \subseteq [\![Proj_{I \cup O}(G)]\!]$, $r' = r[\mathcal{X} \leftarrow 0]$

U2 $(\ell, r) \xrightarrow{G' \wedge G_{r'}, Id_O, \emptyset} (\ell, r')$ *with* $G' = \neg(\bigvee_{G \in Gds_{\mathcal{A}}(\ell, r)} Proj_X(G))$, $r' = I\_Succ(r)$

Our backward reachability algorithm for deterministic VDTA works on $RG(\mathcal{A})$ instead of $[\![\mathcal{A}]\!]$. It computes *predecessors* from which we can reach the target *configuration* of $RG(\mathcal{A})$. A configuration of $RG(\mathcal{A})$ is of the form (q, G) where

q is a state of $RG(\mathcal{A})$ and G is a constraint of $\mathcal{G}(I,O)$. We consider *urgent abstract predecessors* ($aPred_u$), *time-elapsing abstract predecessor* ($aPred_\delta$) and *input-update abstract predecessors* ($aPred_e$) defined over as follows:

$$aPred_u(q,G) = \{(q', Proj_X(G') \wedge Proj_{Var(a)}(G) \mid q' \xrightarrow{G',a,Y}_{U_1} q \\ \wedge Proj_{I\cup X}(G')[a] \subseteq Proj_I(G)\}$$

$$aPred_\delta(q,G) = (q,G) \cup \{(q', Proj_X(G') \wedge G \mid q' \xrightarrow{G',Id_O,\emptyset}_{U_2} q\}$$

$$aPred_e(q,G) = (q, (\neg \bigvee_{G'\in Gds_{\mathcal{A}}(q)} Proj_X(G')) \wedge Proj_I(G))$$

We consider $aPred(Q,G) = aPred_u(Q,G) \cup aPred_\delta(Q,G) \cup aPred_e(Q,G)$ where $aPred_\theta(Q,G) = \bigcup_{q\in Q} aPred_\theta(q,G)$ for $\theta \in \{u,i,\delta\}$ and a set of configurations Q. Finally, $CoReach_a(Q,G) = \mu X.(Q,G) \cup aPred(X)$.

Proposition 1. *Given $q = (l,[x])$ and $G \in \mathcal{G}(I,O)$, $CoReach_a(q,G)$ is effectively computable.*

$\mathcal{Q} = L \times Reg(X) \times \mathcal{C}_M(I,O)$ where $\mathcal{C}_M(I,O)$ denotes the set of constraints on input/outputs the maximal constant occuring in them is lower or equal to M, is finite and $aPred : 2^{\mathcal{Q}} \to 2^{\mathcal{Q}}$ is monotonic. Using fixpoint computation results in [8], we get the termination of the computation of $CoReach_a$. Proposition 2 allows to use $CoReach_a$ in $RG(\mathcal{A})$ instead of $CoReach$ in $[\![\mathcal{A}]\!]$.

Proposition 2. *Let $G' \in \mathcal{G}(I,O)$ be a constraint. It holds that*
$(l,i,o,x) \in Coreach(l', G' \wedge [x'])$ *iff* $((l,[x]),i,o) \in Coreach_a((l',[x']),G')$

Note that *Zones* [4] can also be used for the analysis of VDTA provided a sophisticate construction to handle the urgency in the model. For practical issues zones are more suitable but regions are adequate for a theoretical issues.

3 Automatic test generation

In the sequel, specifications are deterministic. The test selection is based on *Test Purposes* (TP) that allow to select behaviors to be tested from the specification.

Definition 5. *A* test purpose *TP of a specification $\mathcal{A} = \langle L, X, I, O, l^0, G^0, \Delta_{\mathcal{A}}\rangle$ is a deterministic VDTA $\langle S, X\cup X', I, O, s^0, G^0, \Delta_{TP}\rangle$ such that: S is a finite set of locations with a special trap location $Accept_{TP} \in S$, s^0 is the initial location; I, O and X are respectively the input, output and clock variables of the specification; the* initial condition *$G^0 \in \mathcal{G}(I,O)$ is the one of $\mathcal{A}$; X' is the set of clocks with $X' \cap X = \emptyset$, the transition relation[3] is $\Delta_{TP} \subseteq S \times \mathcal{G}(I,O,X,X') \times Id_O \times 2^{X'} \times S$.*

TP is non intrusive with respect to the specification: it does not reset clocks of the specification S and does not assign new values to the output variables. Test purposes are complete, meaning that whatever is the observation of variables or clocks either a transition is taken or the current location does not change. The derivation of test cases proceeds in two steps:

Step 1. Product of the specification $\mathcal{A}$ and the test purpose TP

[3] $G \in \mathcal{G}(I,O,X,X')$ if G is a combination of elements of $\mathcal{G}(I)$, $\mathcal{G}(O)$, $\mathcal{G}(X)$ and $\mathcal{G}(X')$

Definition 6 (Synchronous product). *Given* $\mathcal{A} = \langle L, X, I, O, l^0, G^0, \Delta_{\mathcal{A}} \rangle$ *a specification and a test purpose* $TP = \langle S, X' \cup X, I, O, s^0, G^0, \Delta_{TP} \rangle$*, the synchronous product of* $\mathcal{A}$ *and* TP *is the VDTA* $\mathcal{A} \times TP = \langle L \times S, X \cup X', I, O, (l^0, s^0), G^0, \Delta_{\mathcal{A} \times TP} \rangle$ *where* $\Delta_{\mathcal{A} \times TP}$ *is defined by the following rules (*$\mathcal{R}_1$*,* $\mathcal{R}_2$*,* $\mathcal{R}_3$*):*

$\mathcal{R}_1$*:* $(l, s) \xrightarrow{G \wedge G_s, A, \mathcal{X}} (l', s)$ *with* $G_s = \bigwedge_{G' \in G_{TP}(s)} \neg G'$ *iff there is* $l \xrightarrow{G, A, \mathcal{X}} l'$

$\mathcal{R}_2$*:* $(l, s) \xrightarrow{G_l \wedge G', Id_O, \mathcal{X}'} (l, s')$ *with* $G_l = \bigwedge_{G' \in G_{\mathcal{A}}(l)} \neg G'$ *iff there is* $s \xrightarrow{G, Id_O, \mathcal{X}'} s'$

$\mathcal{R}_3$*:* $(l, s) \xrightarrow{G \wedge G', Id_O, \mathcal{X} \cup \mathcal{X}'} (l, s')$ *iff there are* $l \xrightarrow{G, A, \mathcal{X}} l'$ *and* $s \xrightarrow{G', Id_O, \mathcal{X}'} s'$

An output-update transition can be performed either exclusively in the specification ($\mathcal{R}_1$), or exclusively in the test purpose ($\mathcal{R}_2$), or simultaneously in both systems ($\mathcal{R}_3$). We can show that $Tr(\mathcal{A}) = Tr(\mathcal{A} \times TP)$ and $ObsTr(\mathcal{A}) = ObsTr(\mathcal{A} \times TP)$. The set of locations of the form $(l, Accept_{TP})$ is denoted Acc.

Step 2. Symbolic test case selection and execution. Let $Pass$ be the set of locations of the form (Acc, r) in $RG(\mathcal{A} \times TP)$. Locations of $RG(\mathcal{A}_{TP})$ are tagged with adequate constraints on the input/output variables when computing $CoReach_a(Pass)$. A *symbolic test case* is a path from the initial location of $RG(\mathcal{A} \times TP)$ to a location in $Pass$. The execution of a test case consists, in each location, to select an input or a delay that satifies the tagged input constraint or allows to change the region. An implementation Imp **passes** a test case if and only if any test run leads to a *pass* verdict. Similar to [7], we can show that our algorithm for a specification $\mathcal{A}$ and for all test purposes TP is complete w.r.t. all possible test cases and the relation *tvco*.

4 Concluding Remarks

We described an adapted conformance relation and a model-based testing framework with test purposes for reactive systems modeled by VDTA. A long version of this paper is available in [6].

References

1. Alur, R., Dang, T., Ivancic, F.: Reachability analysis of hybrid systems via predicate abstraction. In: Hybrid Systems: Computation and Control, LNCS 2289. pp. 35–48 (2002)
2. Alur, R., Dill, D.: A theory of timed automata. Theoretical Computer Science 126, 183–235 (1994)
3. Barbuti, R., Tesei, L.: Timed automata with urgent transitions. Acta Inf. 40(5), 317–347 (2004)
4. Bengtsson, J., Yi, W.: Timed automata: Semantics, algorithm and tools. In: Lectures on Concurrency and Petri Nets. pp. 87–124. LNCS (2004)
5. Krichen, M., Tripakis, S.: Conformance testing for real-time systems. Formal Methods in System Design 34(3), 238 – 304 (2009)
6. Nguena Timo, O., Marchand, H., Rollet, A.: Automatic test generation for data-flow reactive systems modeled by variable driven timed automata. Research Report hal-00383203, HAL, CNRS, France (2010)
7. Nguena Timo, O., Rollet, A.: Conformance testing of variable driven automata. In: 8th IEEE WFCS 2010, Nancy, France (May 2010)
8. Tarski, A.: A lattice theoretical fixpoint theorem and its applications. Pacific Journal of Mathematics 5, 285–309 (1955)
9. Tretmans, J.: Test generation with inputs, outputs, and repetitive quiescence. Software-Concepts and Tools 17, 103–120 (1996)

Testing Continuous Systems Conformance Using Cross Correlation

Jacob Palczynski, Carsten Weise, and Stefan Kowalewski

Informatik 11, RWTH Aachen University, Ahornstr. 55, 52074 Aachen, Germany
{palczynski, weise, kowalewski}@embedded.rwth-aachen.de

Abstract. In the development of software for control systems, code is often generated based on models from tools like Simulink. A central question for this approach is in how far model and implementation exhibit the same behavior up to tolerable differences. As this behavior is described by continuous signals, the choice of the best relation describing such an identity is not obvious. In this paper we discuss how to detect similar signal properties shifted in time by analytically calculating the cross correlation of piecewise interpolated sampled signals.

1 Introduction

In the development of software control systems, models and simulations of both the physical system (plant) and the software control system (controller) are used extensively, a prominent tool being Matlab/Simulink [1]. The importance of testing embedded software is a widely discussed topic, in particular their peculiarities, e.g. continuous variables of the controlled systems, or automatically generated code from models [1–3]. As pointed out in [4], models and generated code can behave slightly different in regard to, e.g. continuous variables' value or unwanted time shifts. The question arises to which extend differences in continuous behavior are tolerable, and when signals are to be treated as different.

In our back-to-back-test approach, we treat both model and generated code as black-boxes, i.e. we base our notion of similarity on the observable behaviour and do not consider the inner structure. Instead, we try to compare in how far the implementation behaves similar to the original model up to tolerable differences. These can be slight differences in the actual values, or time-shifts in the signals.The question arises how to compare continuous or at least quasi-continuous signals originating from simulations. As suggested in [5], we define and identify mathematical characteristics in the signals and evaluate the differences between them. In existing work, the evaluation of differences is based on pointwise equality [5, 6, 3].

The problem of tolerable differences can be addressed on different levels of abstraction. In this paper we focus on the recognition of signal properties shifted in time, where we base our comparison on the exact simulation data. We present an intermediate result of our work still in progress, with which we evaluate the time shift between a property in signals by calculating the cross correlation of interpolated signals analytically.

Short Papers of the 22nd IFIP ICTSS, Alexandre Petrenko, Adenilso Simao, Jose Carlos Maldonado (eds.), Nov. 08-10, 2010, Natal, Brazil.

In the following section we briefly present the background of the methods we are to apply, including related work. We discuss our approach in Section 3, followed by some exemplary results of our work. We conclude by showing in which direction we are about to continue our efforts.

2 Background

Modelling tools used in embedded system development, e.g. Dymola, Simulink, use so-called solvers to simulate the models' behaviour. As the name indicates, a solver solves the differential equations representing the model, and chooses the adequate points on the time scale to do so. In this paper, we use Simulink as an exemplary modelling tool. Although the variables in Simulink are treated as continuous over time, the variable data becomes sampled by this type of execution. Depending on the solver, the sampling rate is either fixed or variable.[1]

There is a need of safeguarding that code and original model behave similarly. The idea of comparing signal properties is discussed in [5, 6], but usually the signals are compared on pointwise equality. Little is said on methods that judge if shifts between function values are acceptable. This is where we contribute with this work, which is also a continuation of our efforts so far [7].

In order to detect and recognise potentially time-shifted properties in both signals it is crucial to compare data at the same points in time. We decided to resample the signals by interpolating the signal between the originally sampled data. The existing interpolation methods can be divided in two groups: (i) methods filling up the data between the given points with one function for all intervals, (ii) piecewise methods using one function per interval. We examine several methods with regard to the properties we expect in the following section.

Cross correlation is a well-known method for comparing statistical series, analysing signals, etc., where it is used to evaluate the similarity of shifted data. In spite of that, we are not aware of any application of cross correlation on comparison of simulation and test results in the domain of control systems.

For the computation of the cross correlation of two continuous signals, their functions are shifted by τ, the product is integrated afterwards:

$$R_{xy}(\tau) = \lim_{T_F \to \infty} \frac{1}{T_F} \underbrace{\int_{-T_F/2}^{T_F/2} x^*(t)\, y(t-\tau)\; dt}_{=I} \tag{1}$$

Cross correlation of discrete series is usually calculated in a different way. For example, Matlab offers the function `xcorr` for the cross correlation of two arrays:

$$\hat{R}_{xy}(m) = \begin{cases} \sum_{n=0}^{N-m-1} x_{n+m} y_n^* & m \geq 0 \\ \hat{R}_{yx}^*(-m) & m < 0 \end{cases} \tag{2}$$

[1] Further information on solvers and other Matlab-related topics: http://www.mathworks.com/access/helpdesk/help/helpdesk.html

As we can see in (2), `xcorr` shifts the series by their position in an array, denoted by m. Data sampled at a non-equidistant sample rate is not shifted by a known interval of time τ, instead every point's shift depends on the time differences chosen by the solver. In Section 4, we show that this leads to incorrect results.

3 Comparing Time-shifted Signal Properties with `axcorr`

Our approach consists of two steps, namely interpolating missing signal values for equidistant sampling, and computing the cross correlation of these interpolated signals. Starting with polynomial interpolation, we examine several interpolation methods if they suit our needs. Amongst others we need an interpolation technique which can deal with singular jumps, non-periodic phenomena, and non-zero signals of finite length. A special case are so-called step responses, since often one half part is interpolated well, while in the other only the data points themselves lie on the resulting function. The step response is commonly used to test physical systems where the system's input changes from constant 0 to $x \neq 0$ at a given point of time.

Whereas interpolation techniques which interpolate the whole signal with the same function can achieve good results in the center of the signal, they tend to be unstable on the borders, e.g. high degree polynomials, sums of sine functions, etc. Piecewise interpolation methods prove to be closer to the original signal (Fig. 1), thus being better suited for our purposes. Piecewise linear interpolation is the technique that is easiest to apply and provides decent results, while piecewise Hermite polynomials and cubic splines are more complicated but provide better results with respect to continuity of the resulting function. For these reasons we interpolate the signal data with piecewise methods, for the beginning we use piecewise linear interpolation.

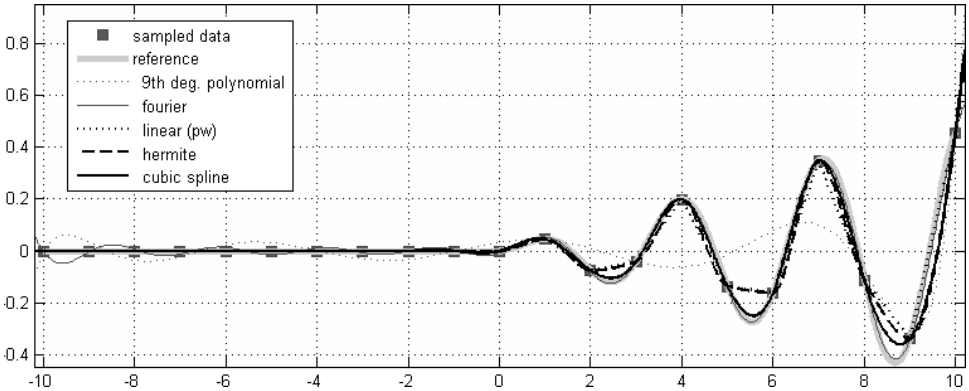


Fig. 1. Comparison of several interpolation techniques. Sampled data origins in a signal switching from constant 0 to $\frac{t}{20}\sin t$ at $t = 0$.

The interpolated data of the signals obtained from both model and code is used to compute the cross correlation. We do not use the `xcorr` function provided by Matlab, but implemented a new algorithm using an analytical solution for the cross correlation integral I in (1). We make use of the fact that our functions are piecewise polynomials and that on each interval between two successive data points their coefficients stay constant. For each value of τ we compute an increasing sequence of points of time $\mathfrak{T}(\tau) = (\mathfrak{t}_0, \ldots)$, whose elements origin from the union of both signals' sets of points of time $\mathcal{T}_x$ and $\mathcal{T}_y(\tau) = \mathcal{T}_y + \tau$, $\mathcal{T}_y(\tau)$ being the shifted version of $\mathcal{T}_y$. We limit $\mathfrak{T}(\tau)$ by the smallest and largest element $\mathcal{T}_x$ and $\mathcal{T}_y(\tau)$ have in common, the difference of these defines T_F in (1) which serves as normalisation factor. By this, the polynomials' coefficients remain constant in the further computation, $x_i(t) = x_{\mathfrak{t}_l,i}$ for $t \in [\mathfrak{t}_l, \mathfrak{t}_{l+1}]$ ($y_{\mathfrak{t}_l,j}$ is defined analogously). We use the sequence $\mathfrak{T}(\tau)$ to subdivide the integral I from (1) in following way:

$$\begin{aligned} I &= \sum_{l=0}^{|\mathfrak{T}|-1} \int_{\mathfrak{t}_l}^{\mathfrak{t}_{l+1}} \sum_{i,j=0}^{n} x_i(t)\, t^i \cdot y_j(t-\tau)\,(t-\tau)^j \; dt \\ &= \sum_{l=0}^{|\mathfrak{T}|-1} \sum_{i,j=0}^{n} x_{\mathfrak{t}_l,i} y_{\mathfrak{t}_l-\tau,j} \sum_{k=0}^{j} \frac{(-1)^j}{i+k+1} \binom{j}{k} \mathfrak{d}_l^{i+k+1} \tau^{j-k} \quad \text{with } \mathfrak{d}_l = \mathfrak{t}_{l+1} - \mathfrak{t}_l \end{aligned} \tag{3}$$

In the end we obtain a polynomial of τ with the same order n as the original interpolation polynomial.

$$R_{xy}(\tau) = \frac{1}{T_F} \sum_{l=0}^{|\mathfrak{T}|-1} \sum_{i,j=0}^{n} \sum_{k=0}^{j} \frac{(-1)^j}{i+k+1} \binom{j}{k} x_{\mathfrak{t}_l,i} y_{\mathfrak{t}_l-\tau,j} \mathfrak{d}_l^{i+k+1} \tau^{j-k} \tag{4}$$

$$= \frac{1}{T_F} \sum_{l=0}^{|\mathfrak{T}|-1} \sum_{s=0}^{n} \left(\sum_{r=s}^{n} (-1)^r \sum_{i=0}^{n} \frac{\mathfrak{d}_l^{i+r-s+1}}{i+r-s+1} x_{\mathfrak{t}_l,i} y_{\mathfrak{t}_l-\tau,r} \right) \tau^s \tag{5}$$

Since our signals have finite length and T_F is finite, too, the calculation of the limit can be omitted. With (5) at hand, it is easy to implement the `axcorr`-algorithm (analytical *cross cor*relation) in Matlab to compute the cross correlation of two continuous signals.

4 Results

To demonstrate the benefit of `axcorr`, we examine the response of a system with PT2 characteristics on a step input. PT2 systems are often found within control systems, a typical example being a circuit consisting of a resistor, a capacitor and an inductor.

In our example we first simulate a PT2-system with a variable rate. In the second simulation we add a delay, which causes the response to be shifted by two seconds in comparison to the first simulation. Additionally the sets of sample

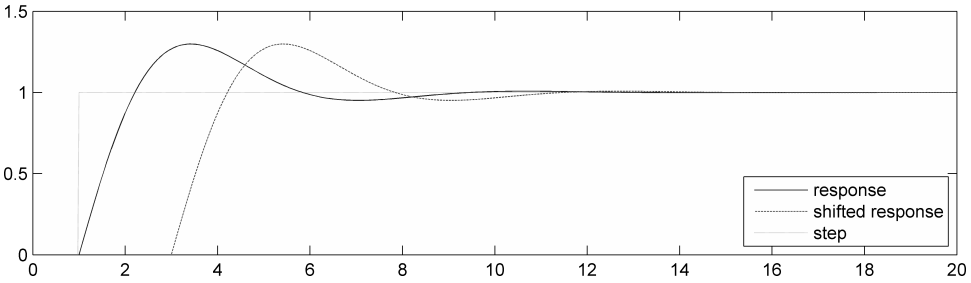


Fig. 2. PT2 step response.

points of time differ, i.e. both signals are sampled at different points of time. Both step responses are shown in Figure 2, where one easily recognises the shift.

Figure 3 compares the results of the different methods to compute the cross correlation of both signals. In order to obtain comparable results we ran `xcorr` with the `'unbiased'` option, which invokes a normalisation similar to the one of `axcorr`. One would expect a maximum at -2, which appears only with the computation with `axcorr`.

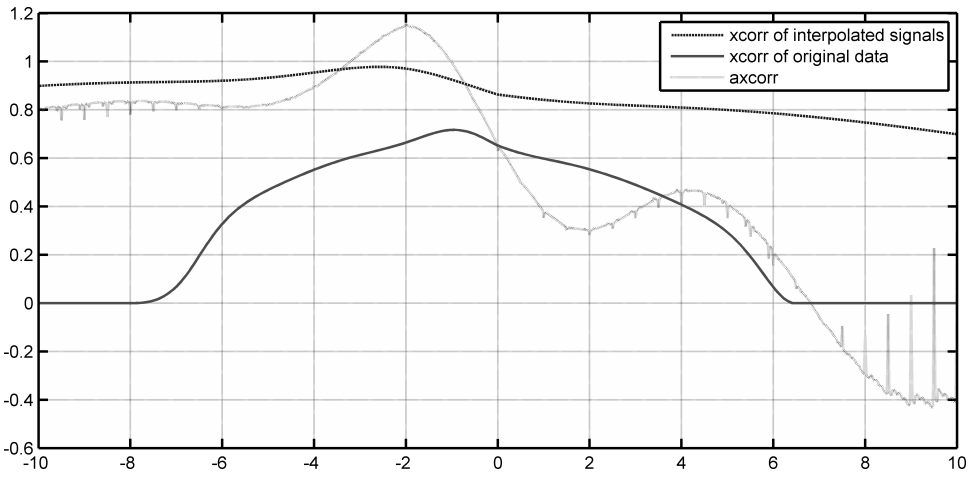


Fig. 3. Cross correlation of PT2 step responses in figure 2.

Both resulting functions for the cross correlation obtained with Matlab's `xcorr` have their maxima for a different values of τ, additionally the peak values are not as distinct as in the results of `axcorr`. The peaks, which appear in the analytical computation near the large absolute values of τ, have their origin in numerical instabilities. We obtained results of similar quality from simulating other systems and other behaviour, e.g. shifted signals with additional noise, and the comparison of a sine and a rectangular signal.

5 Summary

In this work, we present an approach for dealing with signals shifted in time obtained from simulations used for tests. We focus on the question how to treat continuous or quasi-continuous signals sampled with a variable sample rate, where Matlab/Simulink is used as tool to yield simulation data and to implement our approach. To compare the similarity of shifted signal properties we suggest a two step approach, (i) interpolation to resample the signal and (ii) the analytical computation of cross correlation, `axcorr`. Both steps improve the results in comparison to the results one obtains by applying Matlab's `xcorr` function to the original sampled data.

Still, the results can be improved further. On the one hand, up to now we use linear interpolation and expect that cubic splines or piecewise Hermite polynomials will provide better resampled data for cross correlation. On the other hand, when working with Matlab, one has to deal with numerical problems; improvement of our results is possible at this point, too, as we briefly discussed earlier. The next step will be the extension of the analytical cross correlation technique on signals containing more than one signal characteristic shifted individually by different amounts of time. With this on hand, we will integrate this method in our framework for using signal properties for test evaluation.

Acknowledgments This work has been partially funded by the UMIC Research Centre at RWTH Aachen University.

References

1. Stürmer, I., Weinberg, D., Conrad, M.: Overview of existing safeguarding techniques for automatically generated code. SIGSOFT Softw. Eng. Notes **30**(4) (2005) 1–6
2. Stürmer, I., Conrad, M.: Test suite design for code generation tools. In: Proc. 18th IEEE International Conference on Automated Software Engineering. (2003) 286–290
3. Zander-Nowicka, J., Pérez, A.M., Schieferdecker, I.: From functional requirements through test evaluation design to automatic test data patterns retrieval - a concept for testing of software dedicated for hybrid embedded systems. In: Software Engineering Research and Practice. (2007) 347–353
4. Pehinschi, M.G., Gururajan, S., Cukic, B., Napolitano, M.: Investigation of issues in the conversion of simulink models to ansi c code for a neuronal network based adaptive control system. Technical report, West Virginia University / NASA IV&V (December 2005)
5. Großmann, J., Schieferdecker, I., Wiesbrock, H.W.: Modeling property based stream templates with TTCN-3. In Suzuki, K., Higashino, T., Ulrich, A., Hasegawa, T., eds.: TestCom/FATES. Volume 5047 of LNCS., Springer (2008) 70–85
6. Schieferdecker, I., Grossmann, J.: Testing hybrid control systems with TTCN-3: an overview on continuous TTCN-3. Int. J. Softw. Tools Technol. Transf. **10**(4) (2008) 383–400
7. Palczynski, J., Kowalewski, S.: Early behaviour modelling for control systems. In: UKSIM European Symposium on Computer Modeling and Simulation. Volume 0., Los Alamitos, CA, USA, IEEE Computer Society (2009) 148–153

Automating Test Case Execution for Real-Time Embedded Systems

Augusto Q. Macedo[1], Wilkerson L. Andrade[1], Diego R. Almeida[1], and Patrícia D. L. Machado[1]

Federal University of Campina Grande (UFCG), Campina Grande, PB, Brazil
{augusto,wilker,diegor,patricia}@dsc.ufcg.edu.br

Abstract. Testing real-time systems brings a number of challenges to be faced from test case definition to execution in a real runtime environment. Particularly, due to time requirements that need to be properly observed so that accurate verdicts can be reached, automatic test case execution is a very hard and error-prone task. The reason is that test case execution can interfere with the flow of execution of the system under test, adding delays that can lead to false positives of failures. This problem can be even harder for embedded systems due to their limitations. This paper presents a solution to support automation of test case execution for real-time embedded systems.

1 Introduction

Real-time systems (RTS) are computer systems whose correct behaviour depends not only on the generated results but also whether the results are generated at the right time-points. Most RTS are developed for specific purposes and are composed of tightly coupled hardware and software integration such as mobile phones, robots, appliances, vehicles, and so on. In this case, these systems are called Real-Time Embedded Systems (RTES) [7]. For these systems, dependability is an important property that demands rigorous application of V & V activities. Particularly, testing RTES poses a number of distinguishing challenges such as the development platform is usually different from the execution platform, and also there are several execution platforms leading to several cross-development environments. Moreover, RTES are usually composed of parallel activities with synchronous and asynchronous events, with different deployment architectures, and resource limitation and time constraints on the execution environment.

RTES requires at least six kinds of testing [4]: 1) software unit level, where software components are tested in isolation; 2) software integration level, where a set of software components are tested together; 3) software validation level, where a software version is tested; 4) system unit level, where a software version is tested inside its execution platform (e.g., a Real-Time Operating System); 5) system integration level, where a set of system units is tested inside a single node of the execution platform; 6) system validation level, where the complete RTES is tested. In any testing level, a formal specification of the intended behaviour is needed to improve quality and confidence on testing results.

Short Papers of the 22nd IFIP ICTSS, Alexandre Petrenko, Adenilso Simao, Jose Carlos Maldonado (eds.), Nov. 08-10, 2010, Natal, Brazil.

The work presented in [3] proposes an initial symbolic model-based testing strategy for RTES based on symbolic transition systems as test model. Moreover, a possible infrastructure to support test execution in an actual real-time environment is presented. Particularly in [3], the symbolic model-based test case generation strategy is designed to run on a real-time operating system named FreeRTOS [9] – a mini-kernel that can be used to develop real-time systems for embedded devices. The objective of this paper is to extend the previous work by presenting a solution to support automation of test case execution for RTES at software and system integration level. The proposed solution is based on automation of conformance test cases specified by the formalism proposed in [3]. In the sequel, Section 2 presents some tools for testing RTES with their limitations. An overview of the solution is shown in Sect. 3. In Sect. 4, an API developed to support the testing process is described. A case study is presented in Sect. 5. Finally, Section 6 presents the concluding remarks and pointers for future work.

2 Testing Real-Time Systems

TTG is an online and off-line test case generator and monitoring tool for RTS that focus on conformance testing based on timed automata specifications [5]. UPPAAL TRON is another real-time conformance testing tool based on Uppaal timed automata specifications that is able to execute generated tests online through an adapter written in C++ or Java [6]. TTG and UPPAAL TRON are designed for testing at software integration level and, since these tools do not execute in a real-time operating system, they do not provide assurance about time execution. Thus, these tools are not suitable for testing at the system level. Moreover, the underlying models only abstract time. In practice, test cases can be real programs with parameters and variables. Thus, powerful models are needed where variables, time and parameters are explicitly modelled and treated in a symbolic way. Another weakness is that they do not address asynchronous events such as interruptions. Interruption testing has been investigated by Andrade and Machado [1, 2] but time requirements were not taken into account.

Rational Test RealTime[1] is another solution for testing and runtime analysis for RTES at the software level. However, it does not provide automatic test case generation. The tool provides a script language to manually specify test cases. So a communication interface is automatically generated with hollow functions to be filled out by the tester. Test drivers can be designed for some execution platforms (e.g. AIX, HP-UX, Linux, Solaris, and Windows) and they communicate with the SUT through message passing making it difficult to test interruptions.

3 A Test Case Execution Process

A testing process is the sequence of actions the tester should follow to validate the system under test (SUT). The model considered in this work is presented in Fig. 1. The process is divided into four well defined steps.

[1] http://www-01.ibm.com/software/awdtools/test/realtime

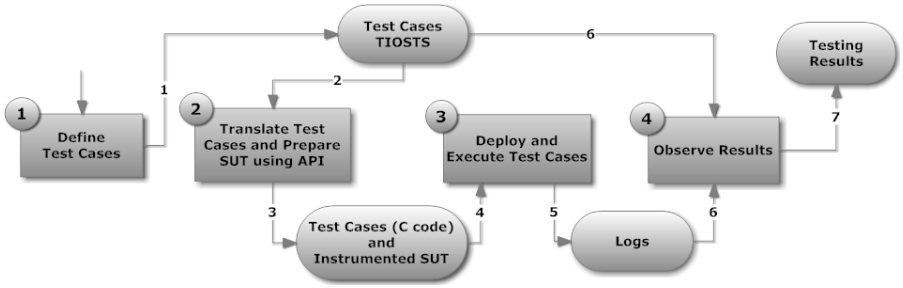


Fig. 1. Testing Process

The first step is to define test cases according to the strategy and theory proposed by Andrade *et al* [3]. This is based on a formalism (named TIOSTS) that is capable of representing the SUT through symbolic transition systems extended with time requirements. From this formalism test cases are automatically generated.

At the second step, each test case is translated into C code. Furthermore, to make the system testable, an API (see Sect. 4) has been developed to instrument the code of the SUT. The code is instrumented for including Points of Control and Observation (PCOs). As the focus of this work is on functional testing, some examples of PCOs are the possibilities of observing values returned by functions, received messages, and timing associated with system responses.

The third step consists in execution of the instrumented SUT guided by test cases. For this, a logging mechanism has been implemented in order to store all information needed to check the testing results. Considering that the SUT runs on a real-time environment such as a real-time operating system, it is important that the implementation logs its own information in order to reduce the number of processes and consequently avoid introduction of noise in the results. As we are dealing with RTES, each addition of code has a direct effect in the execution time of the application. Thus, test case execution can interfere with the flow of execution of the SUT, adding delays that can lead to false positives of failures. For minimizing this interference, all generated information is kept in the main memory. After the SUT execution, a simple text file is generated with the results. Logging frameworks such as log4c[2] are not suitable in this case since the quantity of dependencies forbid their execution within a dedicated hardware and the added code can cause a large delay at the actual execution time of the SUT.

The fourth step receives as input the text file with the execution results and provide verdicts for the tester. This step is not executed inside the execution platform level, but at the development platform level. An extension of the CUnit[3] framework has been developed for evaluating the text file with execution results and emit a verdict according to the test cases defined in the first step.

[2] http://log4c.sourceforge.net
[3] http://cunit.sourceforge.net

4 An API for Implementing Test Cases

This section describes the API developed to support the testing process presented in Sec. 2. The API allows the tester to define PCOs such as observation of values returned by functions and receive messages. Several types are supported such as short, long, double, string, and char. The main contribution of the API is the possibility of checking the time requirements of the SUT.

The developed API is based on the concept of tags which are labels that define specific PCOs. Moreover, some requirements must be met to prepare the testing environment: the SUT must be written in C and its main module must include two libraries in the header (`ActualEventUtil.h` and `ActualEventTester.h`). The usage of the developed API is described by the following steps:

1. The first step is to define tags in order to execute all test cases. The following function is used to initialize the structure of a tag: `int xSetAllTags(char ** tagNames, short quantity);`
2. Next, the following function gives the tester the possibility to set all information to be observed during the test execution: `int xSetTagFields(char * tagName, TagType type, VoidPointer variableToWatch, Boolean shouldWatchTime);`
3. Once the tags of the PCOs were defined, the tester needs to set the points where the information must be caught. Two functions can be used:
   (a) `int xStartWatchingTag(char * tagName)`: it is responsible for putting the tag state into observation mode, getting the initial time of execution, and waiting for a call to the `xEndWatchingTag` function that will finish the watching mode;
   (b) `int xEndWatchingTag(char * tagName)`: it is responsible for finishing the watching mode, catching the final time and the resultant value of the VoidPointer variableToWatch passed by parameter before.
4. The last step is to call the function "`int xLogEventsResult(char ** tagNames, char * logFileName)`" passing a list with all the test case names and the log file name as parameters. This function logs every information at the specified file to be processed at the last activity of the testing process described in Fig. 1.

5 A Case Study

This section presents a case study adapted from [8] and initially discussed in [3]. The SUT is a burglar alarm system whose goal is to monitor sensors to detect the presence of intruders in a building. Consider a test case, presented in [3], that covers the occurrence of an interruption (transfer to the backup power) followed by the detection of an intruder finishing with a call to the police. The objective of the case study presented in this paper is to show how to use the developed API in test case automation (from step 2 to step 4 in Fig. 1). For this, a version of the burglar alarm system was developed to run on the FreeRTOS environment based on industrial PC (x86) port.

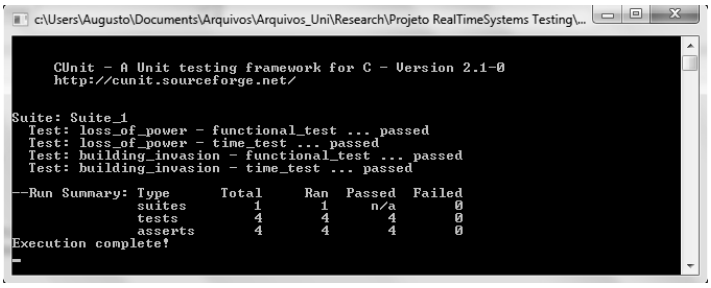


Fig. 2. Testing Results

From Sec. 4, the first two steps are to define the PCOs. The test case of this case study needs two PCOs. One PCO is used to check the interruption caused by the loss of power (tag `loss_of_power`) and another is used to check an invasion exactly after the reactivation of the system (tag `building_invasion`). The tags are defined through the following code:

```
char *pcTags[2] = {"loss_of_power", "building_invasion"};
xSetAllTags(pcTags, 2);
xSetTagFields("loss_of_power", SHORT_TAG, xInterruptionCode, TRUE);
xSetTagFields("building_invasion", STRING_TAG, pcCallResult, TRUE);
```

The third step is to instrument the SUT. The following code shows how the time of an interruption handling is stored:

```
xStartWatchingTag("loss_of_power");
// loss_of_power interruption handling
xEndWatchingTag("loss_of_power");
```

The last step is to call the function responsible for generating the log. As an example, the following code describes the log generated for checking whether the calling was successfully done. Lines 1 and 7 indicate beginning and finishing of an event, respectively. Line 2 has the name of the tag. Lines 3 and 4 presents type and value of the response of the SUT. Lines 5 and 6 indicate initial and final tick of the execution time, respectively. The execution environment considered has a millisecond tick granularity, that is, 1000 ticks per second.

```
1. <BEGIN_EVENT>
2. building_invasion
3. STRING_TAG
4. The call was successfully done!
5. 1
6. 1537
7. <FINISH_EVENT>
```

Considering the testing process (Fig. 1), the generated log is input for the last step where testing results are emitted. For this, the framework CUnit is being extended to compare logs with test cases defined by the TIOSTS notation. Figure 2 presents the initial result considering this case study.

6 Concluding Remarks

This paper presents an ongoing work that addresses a solution to automation of test case execution for RTES at the software and system integration level. The solution provides an API to define PCOs such as the observation of values returned

by functions, received messages, and timing associated with system responses. A case study is also presented to show the applicability of the work. Though it still needs to be extensively validated, the solution addresses the issues on test execution raised in the introduction. It can deal with different development and execution platforms once the SUT is instrumented to run on its target platform and logs are generated to be evaluated at any environment. Moreover, the log generation mechanism allows testing applications with hardware limitations. Environments composed of parallel activities can be tested as the API allows the definition of PCOs for observing message passing and interruptions. The test of time requirements is possible considering a millisecond tick granularity.

Considering the next steps, a tool to support the automatic test case generation based on TIOSTS is being developed. Also, the implementation of the component responsible for observing results (comparison between logs and test case definitions) is being concluded and refactored. Furthermore, there are plans to automate the translation of test cases from the TIOSTS notation to C code increasing the automation of the testing process. The idea is to integrate the results to support the whole testing process shown in Fig. 1 in order to provide a complete environment to generate and execute test cases.

Acknowledgements This work is part of an international cooperation supported by the INRIA (Equipe Associée TReaTiES). This work was supported by the National Institute of Science and Technology for Software Engineering (INES[4]), funded by CNPq, grant 573964/2008-4.

References

1. Andrade, W.L., Machado, P.D.L.: Modeling and testing interruptions in reactive systems using symbolic models. In: SAST'08: Proc. of the 2nd Brazilian Work. on Systematic and Automated Software Testing. pp. 34–43. SBC, Porto Alegre (2008)
2. Andrade, W.L., Machado, P.D.L.: Interruption testing of reactive systems. In: Formal Methods: Foundations and Applications. LNCS, vol. 5902, pp. 37–53. Springer (2009)
3. Andrade, W.L., Machado, P.D.L., Alves, E.L.G., Almeida, D.R.: Test case generation of embedded real-time systems with interruptions for FreeRTOS. In: Formal Methods: Foundations and Applications. LNCS, vol. 5902, pp. 54–69. Springer (2009)
4. Encontre, V.: Testing embedded systems: Do you have the GuTs for it? http://www.ibm.com/developerworks/rational/library/459.html (2003)
5. Krichen, M., Tripakis, S.: Conformance testing for real-time systems. Form. Methods Syst. Des. 34(3), 238–304 (2009)
6. Larsen, K., Mikucionis, M., Nielsen, B.: Online testing of real-time systems using uppaal. In: Formal Approaches to Software Testing. LNCS, vol. 3395, pp. 79–94. Springer (2005)
7. Li, Q., Yao, C.: Real-Time Concepts for Embedded Systems. CMP Books (2003)
8. Sommerville, I.: Software Engineering. Addison-Wesley, 9th edn. (2010)
9. The FreeRTOS.org Project: FreeRTOS. http://www.freertos.org

[4] http://www.ines.org.br

A code based approach to generate functional test scenarios for testing of re-hosted applications

Sharal Nisha Dsouza, Anjaneyulu Pasala, Alexander Rickett and Oscar Estrada

SETLabs, Infosys Technologies Ltd
Bangalore, India
{Sharalnisha_dsouza, Anjaneyulu_pasala}@infosys.com

Abstract. In this paper, we propose a new approach that generates functional test scenarios used to verify the re-hosted software on new environments. This approach systematically extracts the sequence of user-system interactions from source code and these interactions are framed as test scenarios. We have designed and developed a prototype tool based on the proposed approach and case studies have been conducted on an industrial legacy application successfully.

Keywords: Test case generation, software testing, legacy applications.

1 Introduction

It is estimated that the combined value of all business critical COBOL applications running on mainframes is in excess of a trillion dollars. Gartner estimated that there are nearly 180 billion lines of COBOL code in use around the world. Not only the cost of maintaining these legacy applications is increasing but also the organizations are facing difficulty in getting trained people to maintain them. Therefore organizations migrate these applications from mainframe to newer and cheaper maintenance systems based on Microsoft Windows or Linux. This kind of migration is called re-hosting of legacy applications. To facilitate successful re-hosting, tools have been developed that automatically port COBOL source code running on mainframes onto new systems, specifically Microsoft Windows based systems. Micro Focus Studio is one such tool that re-hosts COBOL applications running on mainframes to MS Windows without adding new code or changing existing code manually. However, the re-hosted applications need to be verified for its original functionality on the new systems. Functional test cases are used to verify the correctness of the re-hosted applications. However, being legacy applications, in many instances test cases are not available in any form. Further, due to non-availability of software requirements specifications or design artifacts for these legacy applications, the existing automatic test case generation techniques [1][4][5][12] cannot be used. Therefore testers are facing challenges to test the re-hosted applications. To address these challenges, we propose an approach for generating the functional test scenarios automatically from the application's program code.

Short Papers of the 22nd IFIP ICTSS, Alexandre Petrenko, Adenilso Simao, Jose Carlos Maldonado (eds.), Nov. 08-10, 2010, Natal, Brazil.

The proposed approach characterizes the functional behavior of the application as a collection of sequences of 'unit of behaviors' [1][6]. From the system testing perspective, a 'unit of behavior' consists of a unique combination of a user input, a specified condition on the input data values, and the corresponding predicted response [6]. Therefore, 'one unit of behavior' corresponds to a single test step in a given test scenario. Therefore, a complete sequence of such 'unit of behaviors' for a given specific functionality constitutes one test scenario. The proposed approach captures essentially these set of user input and system responses in the form of <user input, conditions on input values, system output>. Hence, the approach proposes to analyze the application's program code to extract these triplets systematically.

From the application program code, the Abstract Syntax Tree (AST) is constructed by forming independently all the possible paths that can be traversed. The entire path is traversed and all possible set of 'Unit of Behaviors' in that path are extracted. The set of 'Unit of Behaviors' belonging to one path forms a test scenario. Further, to extract the input-conditions-response triplets, the proposed approach involves building a grammar to the programming language specifications and analyzing the program code with the built-in grammar. This is the uniqueness of our code analysis compared to other code based test case generation techniques [8] [9][10][11][12].

Based on our approach, a prototype tool has been designed and developed to analyze CICS based COBOL source code and generates functional test scenarios. Using this prototype tool, case studies on industrial legacy applications have been conducted and results are reported.

2 Related work

Typically three types of techniques have been proposed in the literature to generate test cases. These are (1) Model based, (2) Specification based and (3) Code based techniques.

Model based techniques [1][4][5] takes UML and FSM models built using the requirement specifications as input, and design/architectural information is used to generate test cases. These techniques generate functional tests [1][5] and also integration tests [3]. Ravi etc [1] proposed a technique that presents a relationship between the structure of the UCAD and generated functional test cases. They used the 'Behavioral Slicing using Unit of Behavior' concept in structuring a UCAD similar to the concept introduced in [6]. Several techniques have been proposed to automatically generate test cases from software specifications of the envisaged systems such as Z [7] and ADL [8] etc. These techniques require the functional requirements to be written in a specific formal specification language and are more useful in testing real-time applications. Code-based techniques use code analysis techniques [9][10][11][12] to build the control and data paths and thus generate test cases by traversing these paths. These techniques also use the symbolic execution [9] to generate test cases for glass-box testing. The research being done on code based test case generation techniques is mainly for white-box testing. Basically, test cases for black box testing are generated using the specifications or the model based

approaches. However, these techniques necessitate the software requirements of the application made available either in the form of models or texts. This is not possible with respect to legacy applications.

3 Functional Test Scenarios generation approach

The proposed approach for generation of functional test scenarios is shown in Fig. 1 and broadly has the following activities:

Fig 1: Description of functional tests generation approach

1. *Create* syntax grammar for the complete specifications of the programming language. Generate AST for given application source code. To easily capture the individual paths based on input values, the code is represented in AST. This is a onetime activity for a given programming language.
2. *Build* the Control Flow Graph (CFG) to model user-system interactions. Programmatically every node in AST is analyzed and depending on whether the node is user input, system output or decision statement a corresponding node is created in the CFG. These interactions are in the form of a triplet consisting of user input, conditionals and system output, which precisely a 'unit of behavior' is. The input/output variables from the user screens are retrieved and used in analyzing the nodes of AST.
3. *Generate* test cases from CFG**.** Using depth first search, all independent paths in CFG are identified and extracted. Each independent path consisting of a sequence of <user input, conditionals, system output> is considered as a test scenario**.** These test scenarios form the functional test suite.

```
evaluate true
when eibaid = dfhenter
move  custnoi to cust
evaluate cust
when 0
move "customer does not exist." to messageo
when 1
move  "Name and address" to messageo
move  "LNAME " to messageo
..
```

a)CICS based COBOL code snippet

```
Inqmap1 dfhmdi size=( 24,80), x
.............
initial='customer number'
custno  dfhmdf pos=(5,26), x
attrb=(norm,unprot,ic), x
.................
initial='name and address . . . :'
lname   dfhmdf pos=(7,26), x
attrb=(norm,prot), x
..
```

b)BMS Mapset code snippet

Fig 2. Example source codes

Activities (1) and (2) are described in details in the following sub-sections with CICS based COBOL source code and corresponding BMS map set as example. The COBOL program in Fig 2(a) displays a customer information for a given customer name. If the entered customer number is not valid appropriate error message is

displayed. Fig 2(b) is the BMS mapset for this COBOL code snippet. BMS mapset displays the user screen for the corresponding application that gets launched on the terminal.

3.1 Analyze source code

The analysis of source code to extract user-system interactions involves two steps: (1) Creation of syntax grammar and (2) AST generation.

Creation of syntax grammar. Initially, build the syntax grammar for the complete specifications of a given programming language and generate the parser.

AST generation. The syntax grammar for a given programming language specifications only produces code that recognizes the correctness of the source code written in that language. However, to extract the specific information such as inputs and outputs, AST is used. To generate AST, it has to be specified explicitly while creating the grammar. Therefore, the grammar is required to be augmented with the tree-writing rules which arrange tokens from the lexer, into an AST format. For example, consider COBOL statement, *move "input a number" to message* (means the *message* variable is assigned 'input a number' text). A tree-writing rule *-> ^ (MOVE $src ^ (DST $dst+))* is specified in grammar to generate AST for that statement, where *src* is *"input a number"* and similarly *dst* is *message.*

3.2 Build CFG from the AST generated

To parse AST, a tree-parser is built. The tree-parser extracts input and output interactions to model CFG. The parser has three steps: (1) Extract Input/ Output variables from Graphical User Interface (GUI) code, (2) Create CFG and (3) Identify test scenarios.

Extract Input/ Output variables from GUI code. Typically, user inputs are part of the GUI code such as BMS maps, HTML files etc. Therefore, from the UI source code all input and output variables are extracted using a separate parser. These variables are used in the tree-parser to extract the user-system interactions while parsing the nodes of the AST. The message displayed by the system to the user corresponding to the input/output variable can be obtained from the GUI code. These messages are used to display as text for input and expected output section of the test scenarios being generated. For example, consider BMS file in Fig 3 the code in line numbers 10-12, the message "Customer number" corresponds to the input variable *custno*, hence extracted as an input message.

Create CFG. The AST thus generated and the set of input and output variables obtained from GUI code are input to the tree-parser. In the tree-parsing phase, the statements which are of the type of the user input, conditionals and system output which forms the 'unit of behavior' are alone linked together to form the CFG. Nodes

are extracted by parsing the AST. During parsing the AST, whenever a code corresponding to send information from the user input screen is present, it is considered as user input. Similarly, whenever code corresponding to receive information is present, they are considered as system output respectively.

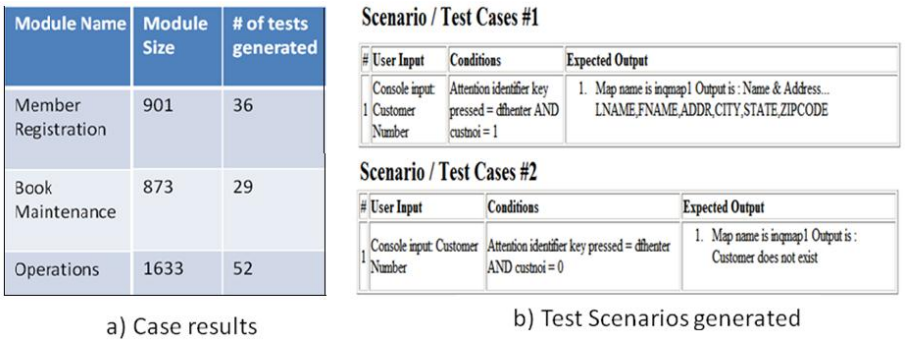
The challenge is to handle the decision nodes that don't directly test the input value. In a typical application, the input variable obtained can be further referenced in the code using the local variables. Their impact also need to be tested and can't be ignored. Hence the indirect reference details (or local variables through which input variables are referenced) are also required to be captured. When the conditional statements test the conditions using these local variables, correspondingly a decision node is created in the CFG.

For instance, consider the code snippet shown in Fig 2.a. In the snippet, the input variable *custnoi* is referenced with the local variable *cust* during the evaluation of the condition. Since the value of the input is tested in the following condition, these decision details also captured. Therefore, a conditional node is added to the CFG. Similarly, many applications contain database interactions that results in more output messages being captured. All these database transactions messages are also required to be verified during the testing by the tester and hence such scenarios also need to be captured. All such the database error messages and successful transactions are captured by creating a decision node correspondingly.

Identify test scenarios. During the construction of AST, for each independent path in the source code a corresponding path in the AST is created. While traversing each path in the AST a START node is added explicitly in the CFG to mark the beginning of the test scenario. A test scenario is a complete path of traversal of an independent path in the AST. The descriptions of the nodes in the AST form the user input, condition and system output nodes of the CFG as explained in the above sub-sections. Whenever there are no more statements in the independent path being traversed or there is an explicit terminating statement in the source code an explicit END node is inserted in the CFG to mark the end of that particular path. Hence a test scenario consist of a set of input, conditions and output nodes starting from START node to END node in the CFG. These test scenarios are generated in XML, HTML and Excel formats with a standard compliance such as HP's Quality Center.

4 Implementation & Case study results

A prototype tool is designed and developed based on the concepts presented here. ANTLR parser generator is used to create syntax grammar for complete CICS based COBOL language specifications. Similarly other parsers are also developed using Java. Using the prototype tool developed, case studies have been conducted on an industrial application. The application is legacy library management system running in one of our client's library. The number of test cases generated for each of the three modules is listed in Fig 3.a. A sample test scenarios generated is shown in Fig 3.b.



Module Name	Module Size	# of tests generated
Member Registration	901	36
Book Maintenance	873	29
Operations	1633	52

a) Case results

Scenario / Test Cases #1

#	User Input	Conditions	Expected Output
1	Console input: Customer Number	Attention identifier key pressed = dfhenter AND custnoi = 1	1. Map name is inqmap1 Output is : Name & Address... LNAME,FNAME,ADDR,CITY,STATE,ZIPCODE

Scenario / Test Cases #2

#	User Input	Conditions	Expected Output
1	Console input: Customer Number	Attention identifier key pressed = dfhenter AND custnoi = 0	1. Map name is inqmap1 Output is : Customer does not exist

b) Test Scenarios generated

Fig 3: Case study resutls

6 Conclusions

We have proposed a new approach to generate test scenarios for verifying the original functionality of the re-hosted applications. The approach is based on capturing and analyzing the structured sequence of user-system interactions from source code. Though the approach is quite generic to be used for applications developed using any programming language, we have built a tool for CICS based COBOL applications. The internal feedback from users of the tool is encouraging.

References

1. Ravi Gorthi and Kailash KP Chanduka, Model-Based Automated Test Case Generation, SETLabs Briefings, Vol. 6, No 1, 2008, pp 39–46.
2. Hartmann Jean, Imoberdorf C., Meisinger M., UML-Based Integration Testing, ACM International Symposium on Software Testing and Analysis, August, 2000, pp 60-70.
3. Basanieri F and Bertolino A, A Practical approach to UML-based Derivation of Integration Tests, 4th International Quality Week Europe, Nov. 20-24, 2000.
4. Lionel Briand and Yvan Labiche, A UML Based Approach to System Testing, Lecture Notes In Computer Science; Springer-Verlag, Vol. 2185, 2001.
5. Wang Linzhang et al., Generating Test Cases from UML Activity Diagram based on Gray-Box Method, Asia-Pacific Software Engineering Conference, Nov-Dec, 2004, pp 284 –291.
6. Gerrard P, Testing Requirements, EuroSTAR, Belgium 1994. www.gerrardconsulting.com
7. Donat M R, Automating formal specification based testing, conference on theory and practice of software development, Lecture notes in CS, Vol. 1214, pp 833-847, 1997.
8. Gargantini A and Heitmeyer C, Using model checking to generate tests from requirements specifications, Lecture notes in computer science, Vol. 1687, pp 146-162, 1999.
9. Cyrille Artho et al, Experiments with test case generation and runtime analysis, Lecture notes in computer science, Vol. 2589, pp 87-108.
10. Elvira Albert et al, Test data generation of bytecode by CLP partial evaluation, Lecture notes in computer science, Vol. 5438, pp 4-23, 2009.
11. Sarfraz Khurshid et al, Generalized symbolic execution for model checking and testing, 9th Int. Con. on Tools and Algorithms for Construction and Analysis of Systems, 2003.
12. Jenny Li J and Howell Yee, Code-coverage guided prioritized test generation, international computer software and applications conference (COMSAC 04), Sep., 2004, pp 178-181.

A tool for automatic generation of executable code from testing models

Fernando Palacios Claudia Pons

LIFIA, Facultad de Informática, Universidad Nacional de La Plata
Buenos Aires, Argentina
{fernando.palacios, cpons}@lifia.info.unlp.edu.ar

Abstract. The UML 2.0 Testing Profile (U2TP) allows the modeling of testing domains, enabling the automatic generation of executable code from those models. This paper presents a proposal to transform structural and behavioral testing models into JUnit code automatically within the context of the Model-Driven Testing (MDT) paradigm. To achieve this goal, a UML 2 Metamodel extension is considered to define a modeling language for testing domains. This language is used to mark the UML diagrams with additional information that is later used to generate the tests cases. In addition, a set of formal rules are introduced to transform testing models into JUnit testing code. Those rules are implemented through a plug-in component for Eclipse, using MOFScript.

Keywords: UML 2.0 Testing Profile, Testing, Testing design, Black-box testing, Models transformation, MOF, MDA, MDT, JUnit.

1 Introduction

As software systems become more complex, the demand for new software development paradigms increases. The Model Driven software Development (MDD) is one of these new paradigms that have emerged to help with the development of high-quality systems. MDD has already proven impact on reducing the time and effort dedicated to the software development process while improving the quality of the final product. This paradigm proposes a process guided by models ranging from the most abstract (Platform Independent Model, PIM) to the most specific (Platform Specific Model, PSM) performing subsequent transformations that lead to the code. In MDD, models are built using a variety of modeling languages, especially UML, which has been proclaimed standard by the OMG [1] and also achieved good acceptance in the software industry.

However, the development of high quality systems requires not only a rigorous development process, but also a systematic testing process to assess the final product. The testing process consists in exercising the program to verify that it meets the requirements and identify differences between the actual behavior and the expected behavior. Thus, the testing process allows the detection of errors and failures in the implementation, guaranteeing the quality of the final product. In the context of MDD, the Model-Driven Testing (MDT) [2] [3] is a kind of black-box test [4] that uses structural and behavior models to automate the test generation process.

Short Papers of the 22nd IFIP ICTSS, Alexandre Petrenko, Adenilso Simao, Jose Carlos Maldonado (eds.), Nov. 08-10, 2010, Natal, Brazil.

Since UML does not provide adequate support for the design and development of tests, the OMG recently promoted an initiative for the creation of a specific language for this purpose. That language was defined with the profile mechanism based on UML and is called the UML 2.0 Testing Profile, U2TP [5]. It allows modelers to design test artefacts and identify the essential concepts of the domain adapted to specific technology platforms. U2TP combines and integrates the development of both the system and the tests via UML. It provides a common communication language for developers and customers and helps developers to reason about the quality and coverage of the tests. This article presents a proposal, consisting of a set of formal rules, to perform automatic generation of executable code from testing models expressed in the U2TP language.

The rest of the paper is organized in the following way; section 2 introduces the U2TP concepts and the platform specific frameworks used in the development of this work. Section 3 describes our proposal to transform U2TP testing models to executable code; section 4 shows a case study. Section 5 describes related works, and finally presents the conclusion and future work

2 Model-Driven Testing Development

In this section we briefly describe the UML 2.0 Testing Profile for testing modeling, and the basic frameworks for test implementation: JUnit [6] and EasyMock [7].

The UML 2 Testing Profile (U2TP) is a modeling language that allows us to design, visualize, specify, analyze, build and document the testing artefacts. This profile provides a rich notation for the specification of testing models under the black-box approach [4]. U2TP is organized into four logical groups covering the following testing features: Test Architecture (structural aspects of a test context), Test Behavior (behavior of the tests), Test Data and Test Time (time constraints and time observations).

The JUnit Testing Framework is an open source framework used to write and run unit tests for applications written in Java [9]. In JUnit, tests are represented as Java classes that define any number of public test methods. JUnit provides a library (*Assert*) to verify the expected against the real result. The test fails if the assert condition is not successfully verified, generating a bug report.

EasyMock is an open source library that extends the JUnit framework by enabling the creation of *Mock Objects* to simulate the behavior of the domain objects. *Mock Objects* are created dynamically, allowing the returning of a specific result for a specific input.

3 Automatic generation of executable tests

Figure 1 shows the main part of the testing modeling language based on the UML Testing Profile. This profile can be applied to structural and behavioral UML2 diagrams representing the system.

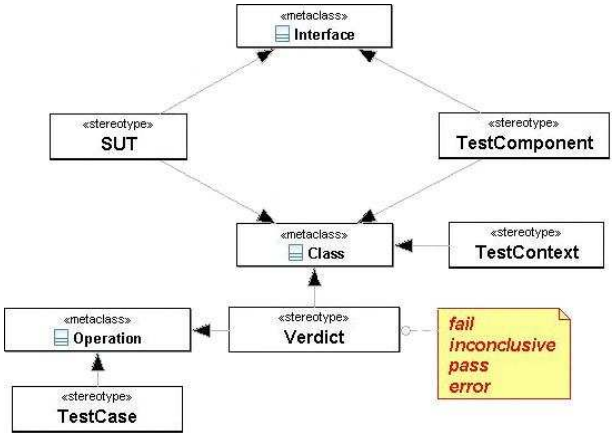


Fig. 1. UML Testing Profile definition

The following rules explain the mapping from platform independent models specified with U2TP to platform specific concepts (i.e., executable JUnit and EasyMock).

Structural concepts are translated as follows:

1. Root classes with *TestContext* stereotype are translated into classes that inherit from *TestCase* in JUnit 3.8; latter versions can use *@Test* annotation. Optionally you can create a constructor with a String type parameter and invoke the parent class constructor passing the String.
2. No root classes with *TestContext* stereotype are translated into classes with the same hierarchy established in the testing model.
3. Operations with *TestCase* stereotype are translated into operations of the class representing the test context. By convention, the name of the operations must begin with "*test*" not having parameters, possible results are:

Pass: the System under Test works as expected.
Fail: there are differences between the expected and actual results.
Error: there is an error (exception) in the test environment.
There is no *inconclusive* Verdict in JUnit; it is usually translated as *fail*.

4. The *setUp()* and *tearDown()* methods can be optionally overwritten in each test class in order to initialize or release resources before or after each test case respectively.
5. Operations with no stereotypes are translated into Java operations of the relevant class.
6. Classes with *TestComponent* stereotype cannot be directly translated into JUnit since it does not support that concept. Thus, we use EasyMock framework as an extension of JUnit to create the test components.
7. Classes with *SUT* (System Under Test) stereotype do not need to be translated; any class in the classpath can be considered a utility or simple class of the *SUT*.
8. On the other hand, the dynamic concepts of the testing models are designed using sequence diagrams. The generation of the testing code from those dynamic models is usually incomplete due to the fact that the models are generally too abstract, lacking the necessary information. Thus, developers are forced to complete that code manually. In this case, EasyMock can be used to complete the behavioral test code.

3.1 An Eclipse Plug-in for the Transformation of testing models into code

Figure 2 shows the architecture of the tool we have implemented as a plug-in for Eclipse [8], on top of EMF [11] and the UML2 plug-in. The functionality of this tool includes U2TP model creation and automatic transformation from U2TP testing models (PIM) to JUnit code (PSM). The transformation is performed by a MOFScript [10] program. The tool is available in [18].

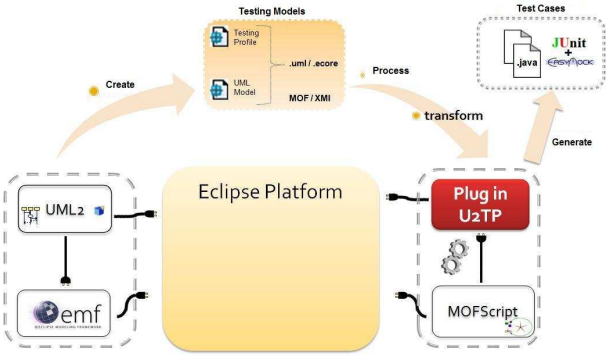


Fig. 2. Deployment architecture.

4 The tool in practice

This section shows how the proposal is put into practice through a use case that performs the authentication and validation of a user. This example shows how the testing stereotypes are applied to structural and behavioral diagrams. Figure 3 shows the class diagram where the following stereotypes were applied on the classes, interfaces, and methods: *TestContext*, *SUT*, *TestCase*, *Verdict*. Then, figure 4 shows an interaction between the objects.

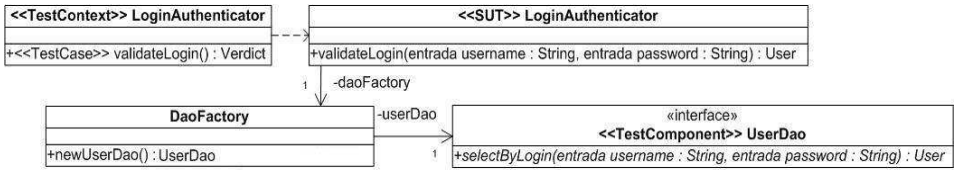


Fig. 3. UML class diagram specified with U2TP.

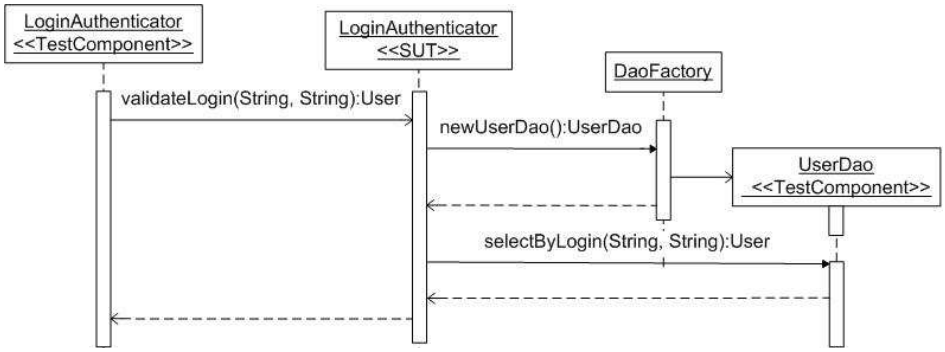


Fig. 4. UML sequence diagram of the test case.

In [18] the reader can find the description of the JUnit code that was automatically generated by the tool. The behavior specified by the sequence diagrams was taken into account by the transformation. For space limitation here we show only the headers of the generated classes:

```
public class LoginAuthenticatorTest extends TestCase
{  public LoginAuthenticatorTest(String name) {super(name);}
  /* (non-Javadoc)@see junit.framework.TestCase#setUp() */
  @Override
  protected void setUp() throws Exception {super.setUp();}
  /* (non-Javadoc) @see junit.framework.TestCase#tearDown() */
  @Override
  protected void tearDown()throwsException{super.tearDown();}
  public void testValidateLogin() throws Exception {
    UserDaoImpl mockUserDaoImpl= …
  }
}
```

5 Related Work, Conclusion and Future Work

Our proposal defines an implementation of the U2TP language to design test cases using UML class and sequence diagrams. Furthermore, our work provides a complete set of formal definitions about the transformation of these annotations into JUnit test code. The test code generation process offers the possibility to generate the skeleton code, enriched with certain kind of behavior specified by sequence diagrams. This behavior generation is achieved by the application of the EasyMock framework. We demonstrate the feasibility of the proposal through the development of an Eclipse plug-in supporting the transformation of U2TP models into JUnit code. The tool was applied to several case studies of medium complexity with satisfactory results.

There are a number of related works. In [13] a code generation tool is presented. Unlike the proposal of this paper, this tool does not use the MDT standards to design testing models. The proposal of transformation through graphical rules described in [14] lacks of tool support to demonstrate its implementation feasibility, besides it does not follow the U2TP standards. The proposal in [15] uses UML 2 and U2TP standard notations to design testing models, also provides tool support through an Eclipse plug-in to generate the skeleton of tests code automatically in TTCN-3 code.

Our work is closely related to the works described in [12] and [16], the latter derived from the first one. Although [12] introduces a methodology prescribing how to apply the U2TP concepts on UML models to get a testing model, the transformation rules are neither complete nor automated by a tool. The research motivation of [16] focuses on the transformation of testing models into executable code in any language, so it defines a UML metamodel extension in OCL, which is then used to achieve the transformations. This work still does not prove such definitions on a software product performing the process automatically.
So far, according to [17], there is no piece of software that assists in the automatic generation of JUnit code based on testing models designed with U2TP. Therefore, a proposal to overcome this problem would be considered very useful as well as a substantial contribution to the MDT field. We plan the following future work based on our results: to extend the tool to support the automatic transformation of other structural and behavioral diagrams additionally to the ones supported so far; to allow transformations of models into another unit testing frameworks; to add more details into the translation process; to support pre and post conditions on test cases and finally to provide the ability to specify the execution sequence of the test cases.

References

1. Object Management Group (OMG). http://www.omg.org
2. Utting, M., Legeard, B., Practical model-based testing, a tools approach. Morgan Kaufmann Publishers, 2006.
3. Mussa M., Ouchani S., Al Sammane W., Hamou-Lhadj A., A Survey of Model-Driven Testing Techniques. In procs of 2009 9th Int. Conference on Quality Software. 2009
4. Beizer, B., Black-Box Testing: Techniques for functional testing of software and systems. John Wiley & Sons, Ltd., 1995
5. UML 2.0 Testing Profile, Final Adopted Specification at OMG (ptc/04-04-02), 2004
6. JUnit testing framework, http://www.junit.org
7. http://www.easymock.org
8. The Eclipse Project. Home Page. Copyright IBM Corp, 2000. http://www.eclipse.org
9. http://java.sun.com
10. Oldevik, Jon. MOFScript User Guide, Version 0.6 (MOFScript v 1.1.11), 2006.
11. Eclipse Modeling Framework EMF. http://www.eclipse.org/modeling/emf/
12. Zhen Ru Dai, Model-Driven Testing with UML 2.0, Fraunhofer FOKUS, Kaiserin-Augusta-Allee 31, 10589 Berlin, Germany. Downloaded on March 2009
13. Boyapati, C., Khurshid, S., Marinov, D., Korat: Automated Testing Based on Java Predicates, MIT Laboratory for Computer Science. Cambridge, MA 02139 USA, 2002
14. Heckel, R., Lohmann, M., Towards Model-Driven Testing, University of Paderborn, Paderborn, Germany, 2003. Downloaded on January 2009.
15. Zander J., Dai Z., Schieferdecker I., Din G., From U2TP Models to Executable Tests with TTCN-3 - An Approach to Model Driven Testing, 2005
16. Lamancha, B., Usaola, M., Piattini, M.:Towards an Automated Testing Framework to Manage Variability Using the U2TP. Procs of the 4th AST, Vancouver. IEEE 2009.
17. Baker P., Dai Z., Grabowski J., Haugen O., Schieferdecker I., Williams C., Model-Driven Testing. Using the UML Testing Profile. Springer-Verlag ISBN 978-3-540-72562-6 (2008)
18. Palacios, Fernando. Master Thesis. University of La Plata. (2010). http://www.lifia.info.unlp.edu.ar/eclipse/pages/tesis_palacios.htm.

Generating Test Cases From B Specifications: An Industrial Case Study

Anamaria Moreira, Ernesto Matos, Fernanda Souza, and Roberta Coelho

Departamento de Informática e Matemática Aplicada
Universidade Federal do Rio Grande do Norte
Natal - RN - Brazil
{anamaria,roberta}@dimap.ufrn.br,ernesto@forall.ufrn.br,fernandamsz@yahoo.com.br
http://www.dimap.ufrn.br

Abstract. In this paper we present a case study in the industry about the application of a test case generation approach based on B machine specifications. First we describe the step-by-step process of the test case generation approach and then we present its use in a particular case study for a door controlling module from a subway system. Lastly, we discuss the results obtained after this study and the future work of our research.

Keywords: B Method, Test Case Generation, Case Study

1 Introduction

The use of formal methods is becoming a common standard in the development of critical systems for providing safe and sound specifications, that can be mathematically verified and proved; and generate code after specification refinements. Even though they bring such benefits, they are not enough to ensure that a system is error free. Thus, testing techniques are considered a complementary practice to formal methods and it is becoming common sense [1] that those two practices, when allied, can result in software that is more safe and reliable.

In this paper we apply an approach for test case generation based on B [2] specifications of a real industrial case. The method was previously defined in [3] and we experimented with it on specifications of a door controlling system module for subways developed by *AeS* (http://www.grupo-aes.com.br). The used specifications can be found in [4]. For more detailed information on this system see [5].

The remainder of the paper is organized as follows: in Section 2 we list and compare related work, in Section 3 we summarize the test case generation approach, in Section 4 we present the door controller specification, its particularities for the case study and the results we have obtained, in Section 5 we discuss our results and in Section 6 we look upon the future of our research.

Short Papers of the 22nd IFIP ICTSS, Alexandre Petrenko, Adenilso Simao, Jose Carlos Maldonado (eds.), Nov. 08-10, 2010, Natal, Brazil.

2 Related Work

During our research we have found three papers that are related to ours [6][7][8]. All of them are using state machine – B or Z – representations to generate test cases, but are limited by the coverage criteria provided, by the kind of test inputs generated, by the limitations of the machines they work with and by the tools they used. Only two types of criteria are provided by these work: equivalent classes and/or boundary values, in our work we add few more criteria based on the use of the orthogonal pairs technique. Another aspect that is improved in our work is the ability to also generate test cases for invalid input data. These work were also limited by the kind of machines they supported, they only worked with machines that had variables classified as integers, sets and/or booleans, in our work we can also deal with interval type variables. Another improvement of our work is the use of more modern tools like the ProB model checker [9]. The ProB tool allowed us to work with machines located in different files and also detect possible breaking of the machine invariants, which was an aspect that was lacking in these previous work.

3 The Test Case Generation Method

A B machine specifies – using set theory, integer arithmetic and first order logic – an abstract state machine. The set of states of the machine are defined by its variables and its invariant, which defines the possible values of these variables. A collection of operations are defined through their preconditions and expected behavior. The applied approach gathers information on global variables and operation parameters from the operation's precondition and from the machine invariant and defines equivalence classes for them. Abstract test data can then be selected after machine animations, combined and refined into concrete test cases. The approach for test case generation consists of the steps illustrated in Fig. 1 and detailed below.

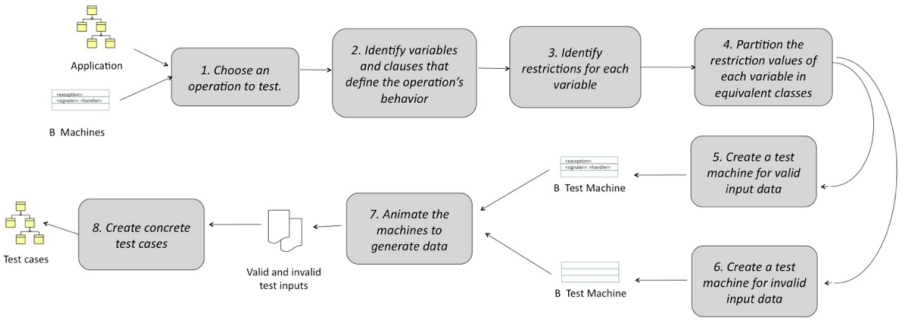


Fig. 1. The approach's overview

- ***Step 1: Choose an operation to test.*** The approach is applied for each operation we want to test, so the first step is to select an operation to generate tests for.
- ***Step 2: Identify variables and clauses that define the operations behavior.*** In the second step we have to look at the variables and invariants of the machine and the parameters and preconditions of the operation we are testing. Two sets are defined: *relevant variables* and *result propositions*. On the *relevant variables* set are placed all the operation's parameters and variables, which occur in the precondition and all the global variables that are directly or indirectly related to them through some restriction on the invariant. The *result propositions* set contains all the operation's precondition clauses and the invariant clauses in which any of the *relevant variables* are restricted.
- ***Step 3: Identify restrictions for each variable.*** In the third step we have to classify all the *relevant variables* according to their type in *intervals*, *belongs to a set*, *sets* or *booleans*. The *classifications* set is created with this information. Besides, the clauses on the *result propositions* set are sort out in two subsets: one containing the *typing clauses* and another containing *non-typing clauses*.
- ***Step 4: Partition the restriction values of each variable in equivalent classes.*** In the forth step we define equivalent classes for the input data of our tests using the equivalence partitioning technique. Based on the *classifications* set we have defined on the previous step we create another set called *formulas* that describe the valid data (valid equivalence classes) for each variable which will be used on our tests. The formulas for each variable are then combined into the *formulas combination* set. Each member of this set is a combination that represents a test case for the operation we are testing.
- ***Step 5: Create a test machine for valid input data.*** To define data that satisfies the restrictions identified for each test case, an auxiliary B machine is created. For each combination of *formulas combination* an operation is inserted in this new machine. These operations have the following structure: each member of *relevant variables* is passed as a parameter of the operation, and the conditions (typing conditions and non-typing conditions) from the combination we are working with are placed as preconditions for the operation.
- ***Step 6: Create a test machine for invalid input data.*** Similarly to the previous step, we define another auxiliary machine to generate data that *do not* satisfy the restrictions on the operation; hence, when we animate it, we will have data for invalid input test cases. We combine the members of *non-typing clauses*, negating one element of each combination. As a result we have the *negative combinations* set and for each of its elements an operation is inserted as we did in step 5.
- ***Step 7: Animate the machines to generate data.*** To generate the data of our test cases we animate the machines created on step 5 and 6 on an animation tool like ProB.

– ***Step 8: Create concrete test cases.*** With the data generated after the machine animations we can implement the concrete tests using a programming language of our choice.

4 Case Study

Our case study uses the B machines specification for a door-controlling module of a real subway system. From the several B machines that specify this module, we have chosen the *General Door Controller* (GDC) machine, which is the main component of the specification.

The GDC machine represents the state of the door-controlling module and defines all the operations and restrictions on these operations. It has operations to open and close the doors on the left and on the right of the subway. Those operations must respect certain preconditions such as the speed of the train at the moment, operation mode of the train, placement of the train in the platform, emergency signals, etc. Bellow we present snippets from the GDC machine:

```
01. MACHINE GDC
02. SEES Sets
03. INCLUDES Opening, Closing, Emergency
04. ABSTRACT_VARIABLES
05. doors_right_open_simultaneously,
06. doors_right_close_simultaneously,
07. internal_speed_over_6km_h,
08. conditions_to_open_satisfied,
09. ...
10. INVARIANT
11. doors_right_open_simultaneously : BOOL &
12. doors_right_close_simultaneously : BOOL &
13. internal_speed_over_6km_h : BOOL &
14. conditions_to_open_satisfied : BOOL &
15. ((internal_speed_over_6km_h = TRUE) =>
16.   (doors_right_open_simultaneously = FALSE &
17.    doors_left_open_simultaneously = FALSE))
18. &
19. ((doors_right_open_simultaneously = TRUE or
20.   doors_left_open_simultaneously = TRUE) =>
21.   (conditions_to_open_satisfied = TRUE))
22. ...
23. INITIALISATION
24. ...
25. OPERATIONS
26. simultaneous_opening_all_doors_right =
27. PRE internal_speed_over_6km_h = FALSE &
28. conditions_to_open_satisfied = TRUE
29. THEN ...
30. END;
31. ...
32. END
```

Fig. 2. Code snippets from the GDC machine

The GDC B specification has nineteen operations, four of them define real behavior of the module like opening and closing doors from left and right, and fifteen of them are only "setters" for the machine state. It has twenty-nine abstract variables, forty-six invariant clauses and each operation has at least one precondition clause.

We have chosen the *simultaneous_opening_all_doors_right* operation to illustrate the proposed approach. After applying the steps 1 to 6 from our approach, we ended up with two auxiliary test machines – one for valid data and another for invalid data – to generate input data for the test cases based on the restrictions on the invariant and the precondition of the operation. The auxiliary test machine for valid data is illustrated on Fig. 3.

```
01. doors_right_open_simultaneously_return,
02. doors_right_close_simultaneously_return,
03. conditions_to_open_satisfied_return,
04. internal_speed_over_6km_h_return,
05. ...
06. <-- simultaneous_opening_all_doors_right_test(
07. doors_right_open_simultaneously,
08. doors_right_close_simultaneously,
09. conditions_to_open_satisfied,
10. internal_speed_over_6km_h,
11. ...) =
12. PRE
13. doors_right_open_simultaneously : BOOL &
14. doors_right_close_simultaneously : BOOL &
15. conditions_to_open_satisfied : BOOL &
16. internal_speed_over_6km_h : BOOL &
17. ((internal_speed_over_6km_h = TRUE) =>
18.    (doors_right_open_simultaneously = FALSE &
19.    doors_left_open_simultaneously = FALSE)) &
20. ...
21. THEN skip();
22. END;
```

Fig. 3. Code snippets from the auxiliary machine for valid data

Using the restrictions gathered from the invariant and the precondition of the operation we are testing, we will be able to generate data that belongs to the given equivalent class defined by these restrictions.

When we animated the machine for valid data (step 7 from our approach), we had to limit the number of generations on the ProB tool otherwise we could have a test case explosion, due to the millions of possible combinations of parameters for the test operation. As a result of the animation we had sets of data from the same equivalent class. To implement the concrete test we only need to select one of these sets, since all would imply the same behavior, since they belong to the same equivalent class. For the invalid data test, we could not animate the test operation because it resulted on a operation with inconsistencies in the precondition. In this case, we could discard this test case since it represented a state that could not happen in the real system

We could not implement and run the concrete tests on the system because it was not implemented yet, but as soon as its development starts, we can implement those test cases. Since its an embedded system, it will be implemented in C and will probably use a test framework like CUnit.

5 Discussions

The case study was extremely helpful for the method validation. The use of equivalent classes derived from the invariants and operation pre-condition clauses reduced the number of test cases from millions to a few in most cases. By using a machine animation tool, we could easily generate input data for operations that used dozens of parameters, which would be difficult to do by hand.

The test cases generated by the approach were considered satisfactory by the *AeS Group* and the proposed method can improve their current testing scenario, by providing a formalized process for test case generation.

Although the approach is usable by the company's current employees, it was considered complex to apply and error prone. Doing each step by hand requires too much attention and if any single mistake is committed the whole process is compromised. Thus, the automation of the approach was considered key to its adoption.

6 Concluding Remarks

In this paper we discussed the application of a test case generation approach based on B machine specifications on a real industrial case study. The next step of our work is the automation of the approach, since it is not a trivial method to apply on a real project and is error prone when done by hand.

We will begin with the first steps, regarding specification interpretation until auxiliary test machines generation (for both valid and invalid data). We will develop a tool that receives as input the B machine specification we intend to generate tests for and passes it through an interpreter, which will extract the data we need to generate the test cases. With the data extracted by the interpreter, a test case generation program will run the approach described in section 3 and output auxiliary test machines for both valid and invalid data tests. From this point further the method will be applied manually. The user will have to animate the test machines on an animation tool, collect the data generated and implement the concrete tests.

Our interpreter will be integrated to the *B-Compiler* toolset, developed by *ClearSy System Engineering* (http://www.clearsy.com), so we can use the technology already developed and widely tested by them in our tool. The adoption of the *B-Compiler* also provides support to the B grammar that is most used on the market.

References

1. Bowen, J., Bogdanov, K., Clark, J., Harman, M., Hierons R., Krause, P.: FORTEST: Formal Methods and Testing, In Proc of 26th Annual International Computer Software and Applications Conference, pp. 91 (2002)
2. Abrial, J. R.: The B-Book: Assigning programs to meanings. Cambridge University Press, New York (1996)
3. Souza, F. M.: Geração de Casos de Teste a partir de Especificações B. Master Thesis, DIMAp/UFRN (2009)
4. Barbosa, H.: Desenvolvendo um sistema crítico através de formalização de requisitos utilizando o método B. Thesis, DCC/UFRN (2010)
5. Déharbe, D., Moreira, A. M., Silva, P. S. M., Russo, A. G.: Modelling Control Systems in B: an Industrial Case Study. In: Brazilian Symposium on Formal Methods Proceedings, pp. 195–197 (2001)
6. Ambert, F., Bouquet, F., Chemin, F., Guenaud, S., Legeard, B., Peureux, F., et al.: BZ-TT: A Tool-Set for Test Generation from Z and B using Constraint Logic Programming. In: Proceedings of the CONCUR'02 Workshop on Formal Approaches to Testing of Software (FATES'02) (2002)
7. Legeard, B., Peureux, F., Utting, M.: Automated boundary testing from Z and B. In: Int. Conf. on Formal Methods Europe, FME02, volume 2391 of LNCS, pp. 21-40 (2002)
8. Bouquet, F., Legeard, B., Peureux, F.: CLPS-B - A constraint solver for B. In: Proceedings of the ETAPS'02 International Conference on Tools and Algorithms for the Construction and Analysis of Systems (2002)
9. Leuschel, M., Butler, M.: ProB: A model-Checker for B. In: FM 2003: 12th International FME Symposium (2003)

Approach for a Real-Time Hardware-in-the-Loop System Based on a Variable Step-Size Simulation

Daniel Ulmer[1] and Steffen Wittel[2]

[1] IT-Designers GmbH, 73730 Esslingen, Germany
daniel.ulmer@it-designers.de
[2] Distributed Systems Engineering GmbH, 73730 Esslingen, Germany
steffen.wittel@distributed-systems.de

Abstract. In contrast to a formal validation, the Run-Time Verification observes the behavior of the System Under Test (SUT) during its execution. For example, automobile manufactures usually perform complex driving maneuvers on Hardware-in-the-Loop (HiL) systems to ensure the correct functionality of their Electronic Control Units (ECUs). Thereby, the currently used HiL systems often complicate the verification process as a result of their workload and the wait times involved after modifications of the SUT. In order to make this process more effective and efficient, the introduced approach is based on an off-the-shelf PC running a variable step-size simulation in combination with an intelligent I/O-device. An evaluation of the prototypical implementation showed processing times, which already satisfy the timing requirements of common ECUs.

1 Introduction

Current Advanced Driver Assistance Systems (ADAS) like DISTRONIC PLUS or Brake Assist PLUS do not only warn the driver of a possible hazard, but also mitigate rear-end collision and head-to-tail crashes autonomously as described by Daimler AG [2]. Hence, automotive manufactures verify such safety-relevant systems elaborately to avoid endangerments of the passengers as well as of other road users. Despite all improvements in the design and implementation, e.g., the automatic code generation that facilitates a consistent quality of the source code as well as compliance to software development standards like MISRA-C [5], error-free implementations still cannot be guaranteed. Therefore, extensive software and hardware tests are conducted in order to find defects in a stage of the development as early as possible and especially not detecting these after the rollout [12].

1.1 Current Situation

MATLAB, Simulink and Stateflow [13] are a common solution [20] for modeling and design of embedded systems. They constitute a de-facto standard in the automotive industry with a market share of approximately 50 % [6]. The solution

Short Papers of the 22nd IFIP ICTSS, Alexandre Petrenko, Adenilso Simao, Jose Carlos Maldonado (eds.), Nov. 08-10, 2010, Natal, Brazil.

is usually used in combination with additional toolboxes such as the Vehicle Network Toolbox [15] from The MathWorks Inc., which allows a direct processing of CAN messages from a hardware interface like the Kvaser Leaf Professional [9] or the Vector CANcaseXL [18]. In this manner, MATLAB can be used to validate Simulink models with live captured data as well as to execute test cases for an externally connected SUT. In principle, this way is rarely applied–especially not to implement test cases for a HiL environment. For HiL testing, other software products like TPT [1] or LABCAR [3] are used with their proprietary notations as well as particular hardware requirements.

1.2 Need for a new Approach

The MATLAB toolchain [13] is not sufficient to act as the base for a HiL system, because the software itself cannot ensure the commonly required real-time behavior on PCs without a Real-Time Operating System (RTOS) [8]. Moreover, different environments [1, 3, 13] are often used for automotive software development and testing. In addition, the workload of HiL systems is typically high due to the purchasing costs [21] for setting up such testing environments as well as the increasing number of ECUs [4] to be verified for one vehicle line. In consequence of this, it can take time until test cases are performed and analyzed that opposes a rapid prototyping process of innovative ADAS.

2 Approach for a Real-Time HiL System

The fundamental idea behind the new approach for a real-time HiL system, which will be further called RTA HIL, is the separation of the calculation intensive parts of the process from the parts that are responsible for the real-time behavior. Thereby, the RTA HIL uses an intelligent I/O-device as illustrated in Fig. 1 with the name Real Time Adapter (RTA) [17] and an off-the-shelf PC. On the one hand, the RTA device provides the necessary hardware interfaces for the SUT and observes the time constraints. On the other hand, the PC executes the test cases within a Discrete Event Simulation (DES) [10] using Simulink and Stateflow.

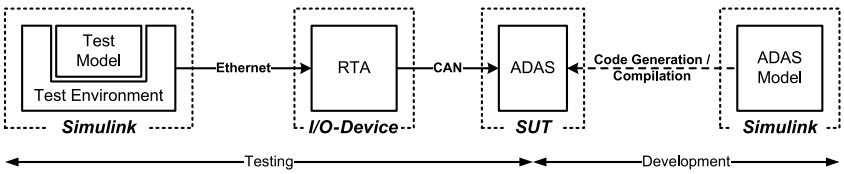


Fig. 1. Development and testing environment

The internal, highly-precise clock of the RTA takes response for the real-time behavior of the whole testing environment. Based on timestamps added to

the messages between the RTA and the PC, the simulation is able to calculate the accurate sending points of the outgoing messages independently from the processing time required for the simulation as shown in the following figure:

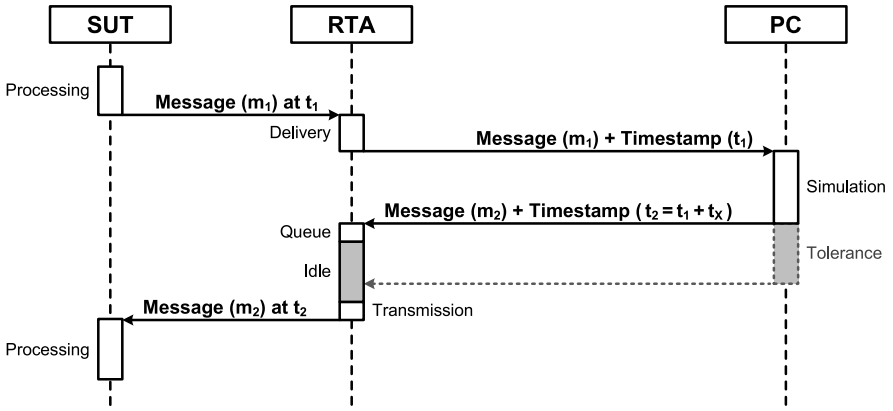


Fig. 2. Processing of incoming and outgoing messages

All outgoing messages received from the simulation are queued by the RTA. The messages are transmitted, if the timestamps and thus the intended sending points match with the time of the internal clock. If the RTA detects a sending point in the past less an adjustable tolerance, it notifies the PC about the time fault. In this case, the PC aborts the current execution and restarts the test case a few times to exclude time violations caused by singular incidents. Notifications are also sent after the expiration of one of the hardware timers available on the RTA, which can be used to realize wait times in the simulation between a stimulus and the expected reaction of the SUT. During the test execution, the simulation is advanced from one event to the next and does not depend on fixed time-steps as usual for a real-time HiL system [11]. This behavior reduces the number of necessary simulation steps and in consequence it decreases the processor workload of the system.

3 Seamless Integration

The RTA HIL uses a so called system-function (S-function) [16] to extend the capabilities of the MATLAB toolchain with additional user functionality and to provide a Graphical User Interface (GUI) for the test cases implementation. In this manner, the extension does not differ from a built-in block of Simulink and can be combined with other blocksets.

The S-function imitates a discrete-event behavior with functions from the Operating System (OS), which typically synchronize threads. Thereby, it suspends

the execution of the innately single-threaded simulation engine of Simulink after each step as displayed below in Fig. 3 until a new event occurs.

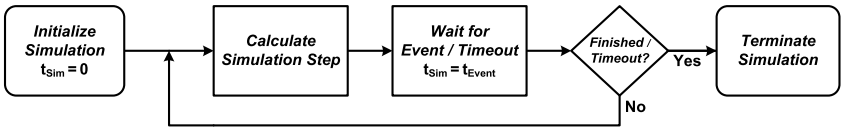


Fig. 3. Imitation of the discrete-event behavior within the S-function

The solution causes difficulties, because other functions, e.g., the processing of the incoming messages from the RTA, are not performed during intervals without events. In the same manner, the GUI of Simulink or Stateflow is not updated and does not accept any input from the user. To get around, additional threads are used in the S-function to handle user-defined activities independently from the execution state of the DES as well as further simulation steps are introduced that only have the task to refresh the GUI as well as to process the user input.

The SUT usually performs its calculations with Fixed-Point Numbers [19], whereas the simulation on the PC applies Floating-Point Numbers [7]. Therefore, the performance-optimized S-function has to convert the signal values between both formats based on the specified data types [14] and scalings that represent the way in which the data is interpreted.

4 Evaluation and Analysis of the Performance

For the evaluation, the processing time is defined as the sum of the time needed for all activities that are performed or caused by the PC and the RTA during the execution of one simulation step. Simplified, it can be said that this is the time between an incoming and the corresponding outgoing message.

The generic measurement setup consists of a common off-the-shelf PC, a RTA device and an evaluation board with two CAN interfaces. The evaluation board represents the SUT and serves as an adjustable traffic generator, whereas the S-function is instrumented to collect and store the timestamps received from the RTA. All options for the optimization in Simulink are switched off to get comparable results during the evaluation and to avoid performance enhancements in advance of the test execution.

A setup with an empty test case, which directly feeds-through the values from the input to the output ports, represents the RTA HIL latency caused by the testing environment. As shown in Fig. 4, an average processing time of 2.2 ms is needed to receive a message from the SUT and response to it immediately without calculation steps.

Test cases comprise not only relative simple logic, but also complex calculations. This raises the question, whether the performance of the RTA HIL system

is sufficient for common ECUs. Therefore, a floating-point multiplication cascade consisting of 1000 Simulink blocks was used in another test model. Figure 4 displays in addition to the RTA HIL latency the measured processing time of this setup with an average value of 3.3 ms.

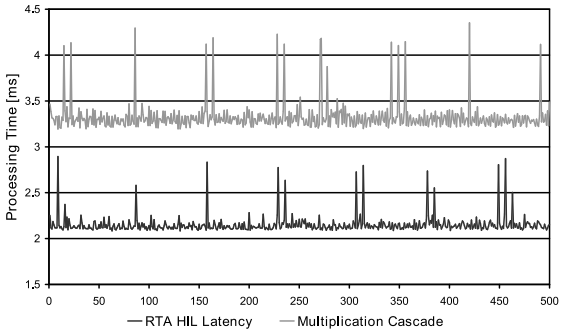


Fig. 4. Results of the performance evaluation

However, some outliers are clearly visible in the graphical representation of both results due to the non-deterministic processing of the PC. A performance reserve in the range of a few milliseconds can mostly compensate the deviation and enables the RTA HIL to produce convincing and reproducible outcomes.

5 Conclusion

The evaluation revealed an average latency of approximately 2.2 ms for an empty test case and about 3.3 ms for a 1000 blocks multiplication cascade. Overall, the performance of the RTA HIL system seems to be sufficient to implement complex test cases with a decently adequate reserve for the compensation of possible outliers. Thereby, time faults are not an exclusion criterion for the approach, because a HiL environment is not a directly safety-relevant system.

The workload of the computer system substantially determines the time needed for the execution of a simulation step and thus for the processing time of the RTA HIL. Therefore, the applied DES reduces the processor utilization significantly by the minimization of the necessary calculations during a test run. A further increasing of the performance is achieved by the execution of the static simulation parts in a binary-optimized form, whereas the frequently changing parts are interpreted directly within Simulink during a test run.

An implementation challenge of the RTA HIL system has been shown up that Simulink runs the GUI and the simulation in the same thread, which in our case prevents the user from an interaction with the testing environment during time periods without events. This behavior is caused by the applied S-function that suspends the thread between two events. Thus, additional simulation steps are necessary in the meantime to process the user-defined inputs and outputs.

6 Further Work

The implementation of the new approach for a real-time HiL system is still under development and therefore the existing hardware and software parts are advanced continuously. One of the pending points constitutes the full integration of the RTA with its timestamps in Simulink that includes, among others, the usage of the device as external time provider.

References

1. Bringmann, E., Krämer, A.: Model-based testing of automotive systems. In: Proceedings of the 1st International Conference on Software Testing, Verification, and Validation (2008)
2. Daimler AG: The challenge of accident prevention. Milestones in Vehicle Safety. The Vision of Accident-free Driving (2009)
3. ETAS GmbH: LABCAR System, http://www.etas.com/en/products/labcar_system_components.php
4. Gerke, T.: The Virtual Vehicle. EE Times (2009)
5. Hatton, L.: Safer C: Developing Software for in High-Integrity and Safety-Critical Systems. McGraw-Hill Companies (1995)
6. Helmerich, A., Koch, N., Mandel, L., Braun, P., Dornbusch, P., Gruler, A., Keil, P., Leisibach, R., Romberg, J., Schtz, B., Wild, T., Wimmel, G.: Study of Worldwide Trends and R&D Programmes in Embedded Systems in View of Maximising the Impact of a Technology Platform in the Area. Tech. rep., FAST GmbH, Technische Universität München (2005)
7. IEEE Computer Society: IEEE Standard 754-2008 for Floating-Point Arithmetic
8. IntervalZero: Hard Real-Time with IntervalZero RTX on the Windows Platform, http://www.intervalzero.com/pdfs/RTXWhitePaper-6-09.pdf
9. Kvaser AB: USBcan Professional, http://www.kvaser.com/index.php?option=com_php&Itemid=258&eaninput=7303130003576
10. Pooch, U., Wall, J.: Discrete Event Simulation: A Practical Approach. CRC (1992)
11. Ramaswamy, D., McGee, R., Sivashankar, S., Deshpande, A., Allen, J., Rzemien, K., Stuart, W.: A Case Study in Hardware-In-the-Loop Testing. SAE Technical Paper Series (2004)
12. Starkloff, E.: What Can Toyota Teach Us About Test? Electronic Design (2010)
13. The MathWorks Inc.: MathWorks Product Overview Brochure
14. The MathWorks Inc.: Simulink User's Guide, Working with Data Types
15. The MathWorks Inc.: Vehicle Network Toolbox User's Guide
16. The MathWorks Inc.: Writing S-Functions, What is a S-Function?
17. Ulmer, D., Theissler, A., Hünlich, K.: PC-Based Measuring and Test System for High-Precision Recording and In-The-Loop-Simulation of Driver Assistance Functions. In: Proceedings of the Embedded World Conference (2010)
18. Vector Informatik GmbH: CANcaseXL, http://www.vector.com/vi_cancase_xl_en.html
19. Yates, R.: Fixed-Point Arithmetic. Tech. rep., Digital Signal Labs (2009)
20. Zander-Nowicka, J.: Model-based Testing of Real-Time Embedded Systems in the Automotive Domain. Ph.D. thesis, Technical University Berlin (2009)
21. Zhao, J., Li, Z., Ding, J., Chen, L., Wang, Q., Lu, Q., Wang, H., Wu, S.: HIL Simulation of Aircraft Thrust Reverser Hydraulic System in Modelica. In: Proceedings of the 7th International Modelica Conference (2009)

Automated GUI Testing on the Android Platform

Martin Kropp, Pamela Morales

Institute of Mobile and Distributed Systems, University of Applied Sciences Northwestern Switzerland, Steinackerstrasse 5,
5210 Windisch, Switzerland
{martin.kropp, pamela.morales}@fhnw.ch

Abstract. Mobile software applications are even more focused on good user experience than traditional desktop applications. Today's high-resolution displays of mobile devices offer increasing capabilities in user interaction experiences. Correct behavior of the emerging GUIs has become a critical success factor. However with the more advanced GUI capabilities, manual testing of the GUI has become more complex and error prone. Thus it is absolutely critical for efficient mobile application development to establish automated GUI testing. Android offers various concepts for testing mobile applications. In this paper we present two approaches for testing mobile GUI applications, the Android Instrumentation Framework and the Positron Framework. We will show the strength and weaknesses of the two approaches.

Keywords: Test Automation, Test Design, Testing Tools, Software Engineering, GUI Testing, Android.

1 Introduction

In software development, an adequate graphical user interface, in general, and with this, its correct behavior and operation has become a critical success factor for the acceptance of GUI-centric applications. This is even more important for applications on mobile devices with their limited display size and their special input devices (like fingers, sticks, or small keyboards). Thus testing the correct functioning of the GUI has become very critical, too. Because of the inherent limitations of manual testing of the GUI, there is a strong need for automated GUI testing. However automated GUI testing for mobile applications is still at its beginning. Google's Android platform provides two approaches: The "Android Instrumentation Framework" and "Positron Framework". The goal of this paper is (1) to show how to write GUI tests with both approaches by example and (2) to analyze the strength and weaknesses of the two approaches and compare them to desktop testing techniques.

After an introduction about Automated GUI Testing in general, we present the Android Instrumentation Framework and the Positron Framework and a comparison of both. A final outlook on further work concludes the paper.

2 Automated GUI Testing

Short Papers of the 22nd IFIP ICTSS, Alexandre Petrenko, Adenilso Simao, Jose Carlos Maldonado (eds.), Nov. 08-10, 2010, Natal, Brazil.

GUI testing is the process of testing an application with a graphical user interface to ensure correct behavior and state of the GUI. This includes verification of data handling, control flows, states and display of windows and dialogs, for example [1].

Testing mobile GUI applications raises special challenges: The event-driven nature of GUI's makes GUI applications non-deterministic; the user can click anywhere on the screen. For mobile applications the development platform is usually different from the target platform. It is impossible to test the almost unlimited numbers of possible states a GUI can have. This makes mobile GUI testing more difficult compared to pure functional testing [2]. Manual GUI testing is very error prone and hardly reproducible, and causes very high effort.

Providing automated GUI testing for mobile applications would improve the situation significantly, including regression testing. The idea in automated GUI testing is to develop testing scripts which simulate user interactions with the GUI application and verify the correct behavior, state and control flow in the GUI to discover possible deviations from the expected behavior. The tests are structured against the GUI application to elucidate and clarify what is required from user perspective.

We use a simple Contact Administration application to store, view and edit a list of contacts as a case study.

3 Android Instrumentation Framework

The Android Instrumentation Framework is an integrated part of the Android software development kit (sdk). *Instrumentation* refers to the ability to monitor and diagnose an application by inserting tracking code, debugging techniques, performance counters, and event logs into the code, which also allow measuring its performance and control its behavior [4].

The Android Instrumentation Framework includes supporting test classes (see **Fig. 1**), which allow to start, run, control and terminate an application in test mode. Depending on the testing level [6] these classes provide facilities for controlling and ensuring the access through the application and its resources [5].

The Android Instrumentation Framework is based on the JUnit framework. The *ActivityInstrumentationTestCase2* extends the JUnit core *TestCase* class. This allows using the standard JUnit assert-functionality for verifying expected and actual behavior in the GUI caused by user interactions, fired events, etc. [4]. This makes using the instrumentation framework very easy for experienced JUnit developers.

In Android each screen of an application is represented by an Android *activity*, so an Android application can be seen as an activity stack. Each activity itself consists of groups of ordered UI elements. Each activity has its independent lifecycle. Hence, each activity can be tested separately. Android instrumentation provides the special class *ActivityInstrumentationTestCase2* for this testing level.

ActivityInstrumentationTestCase2 offers functionality for handling GUI resources and accessing the GUI on the activity context level. The programmer creates a test class derived from *ActivityInstrumentationTestCase2* for each activity and specifies the activity class to be tested (class under test – CUT, **Listing 1**).

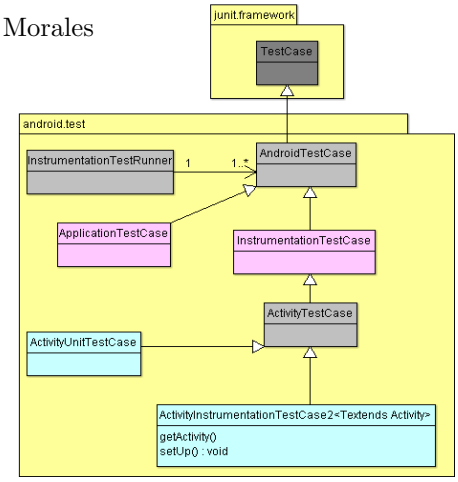


Fig. 1. Android Instrumentation Framework Class Diagram

A GUI test simulates a process related to the UI elements in the activity or a specific expected user behavior [6]. The general structure for an instrumentation test class is shown in **Listing 1.** The listing demonstrates a test adding a new person to the list of contacts and verifying the correct behavior and state of the GUI. The low level API of the Android Instrumentation Framework allows sending key strokes to the selected control using the `sendKeys()` method to simulate specific user interactions; in the sample to enter the surname of a person.

For testing the user action events (in Listing 1. to click "Save" button) the actual test is executed as a Runnable. The overloaded run() method of the Runnable class includes event call and the assertions. The Runnable instance is passed to the UI thread which runs in parallel with the tested activity.

The Instrumentation Framework uses the JUnit assertions to verify the GUI state and behavior. No new verification concepts are introduced, which makes it easy to use for experienced JUnit tester. The tests are compiled and bundled as an independent instance of the application.

```
public TestDemo(String pkg, Class<Demo>activityClass) {
   public TestDemo(){
   super("org.demo", Demo.class);//Bundle activity
}

protected void setUp() throws Exception {
   super.setUp();
   final Demo2 a = getActivity();//Instance Activity

   /*Bundle widgets by id in file R*/
   surname=(TextView)a.findViewById(R.id.surname);
}

public void test1AddPerson (){
   sendKeys( KeyEvent.KEYCODE_T); //Type a letter
   sendKeys( KeyEvent.KEYCODE_O); //Type a letter
   sendKeys( KeyEvent.KEYCODE_O); //Type a letter

   //runnable for save button action event
   Runnable saveRun = new Runnable() {
      public void run() {
         save.performClick();//Simulate click event
         p=getActivity().getPerson();//get data from UI
         save.setEnabled(false);//set UI element status
         Assert.assertFalse(save.isEnabled());
```

```
        }
    };
    getActivity().runOnUiThread(saveRun);//Run test
}
```

Listing 1. Test Class Structure with Android Instrumentation Framework

4 Positron Framework

The Positron framework is a client-server model built on top of the Android Instrumentation Framework to handle the activity's resources, offering a Selenium-like high-level approach for writing and running test cases [7]. Each test case is treated as a client, which connects to the server component that runs the activity [8]. The framework provides the communication infrastructure and the server services [7]. The communication network services use the Android Debug Bridge (adb) to establish the connection between the server and the clients [7]. Each test method (client) creates its own activity instance to communicate with.

The Positron framework is thread-safe, which is especially important to control the UI elements and their events running in a separate thread [7]. To access activity resources it uses a path with dot-separated property notation [8] (see **Listing 2**). It considers the activity as a container of UI elements arranged in a hierarchy. In addition, Positron provides variations of the method *At* to tweak the return type according to the required property, such as: *stringAt, intAt*, for example.

The test class must extend class TestCase. The structure of the test class is similar to the structure of the standard JUnit test class. Also, all verifications about behaviors and data are made by "asserts", which are structured as in JUnit [8]. **Listing 2** shows a typical test class written with Positron and some typical methods to simulate user interactions with the GUI.

Positron includes its own implementation of Selenium-like commands to allow automated run of high-level GUI test suites. Each test class consists of a set of test methods representing user tasks. This enables to run the tests independently of each other [9]. In addition, Positron provides synchronization functionality between the application under test and the tests for concurrent access of GUI elements during the testing scenarios. The negotiation is made by Positron when it opens the connection between server and client, i.e. when the activity is started in test mode [7].

```
public class AddRecord extends TestCase {
@Before
public void runBeforeEveryTest() {
//Start the activity in test mode
startActivity("org.demo.Demo","org.demo.demo2.Demo2");
pause();//Wait for the requested answer
press("Name", DOWN); //Simulate typing a word
click();//Simulate click event in a focused UI element
}
```

```
@Test
public void addPerson() throws InterruptedException {
/*Assert state*/
assertTrue("not click", booleanAt("save.isPressed"));
field1=stringAt("listView.1.0.text");//Get string
}
}
```

Listing 2. Test Class Structure with Positron

5 Framework Comparison

Comparing both approaches in general, the Android Instrumentation Framework is a low-level API to simulate user interactions with the screen (**Listing 1**), while Positron provides an abstracted comfortable high-level interface (**Listing 2**) for writing GUI tests. The Android Instrumentation Framework gives a direct handle to the context used to poke freely around the activity to validate test assertions, and to access the widgets' properties directly in the test case. This ensures efficient runtime and fast response. The Positron Framework connects to the application under test each time when the test class needs to use activity resources which slows down test runtime execution.

With the Android Instrumentation Framework, every UI element must be brought individually in the setup method; i.e., the GUI simulation is a re-implementation of the activity under test (see in Listing 1, setup() method). The run() method of the Runnable class must be overridden methods to handle user action events. Though these approaches allows writing test cases at a much lower level with direct resource access, this leads to much more test code, which makes writing tests more error prone and increases maintenance effort.

Positron provides methods to locate activity resources by calling getters for each named property class. This avoids a rewriting of the activity within the test class (see in Listing 2, setup() method). On the application code side, this requires that the class under test (the activity class) has implemented the corresponding getter methods to the resources. The UI objects' methods are handled by Positron. The developers do not need to take core of the synchronization between the UI thread and UI objects methods, as with the Instrumentation Framework. However, this limits the UI Objects handling at screen level.

While the Android instrumentation framework offers greatest flexibility and direct access to the GUI controls through its low-level API, it also requires writing more test code, which increases error rate and also causes higher maintenance effort. The Positron framework provides a high-level interface for writing automated GUI tests, which reduces the effort, for both, writing and maintaining test code significantly.

The strengths of both frameworks are: use of instrumentation for handling UI resources through the activity; user interactions simulation by sending key events; execution on the target platform; usage of standard JUnit assertions to verify GUI behavior and states.

Among the weaknesses of these approaches are that the tester must have a detailed knowledge of the source code under test to find the UI resources in the code. The

developers cannot write tests spanning multiple activities, because the frameworks isolate the test class for handling only one activity from an activity stack.

6 Conclusion and Outlook

Our analysis of GUI testing tools for mobile application development, exemplified by the Android platform as one of the currently most emerging environments, showed, that this is area is still in its beginning. The two analyzed frameworks provide basic GUI testing functionality on various levels. Compared to GUI desktop testing tools both frameworks yet show notable limitations.

These limitations include: capture/replay functionality, support for GUI controls, script generation, and runtime control in a compulsory interruption environment, and platform neutral testing, i.e. testing of the various OS and runtime platform of the very defragmented device market. In a current research project we are investigating into the question, how a ported mobile application can be enforced to follow a given architecture [10].

References

1. Gerrard Paul, "Testing GUI Applications", EuroSTAR Conference, Edinburgh, November 1997, http://www.gerrardconsulting.com/GUI/TestGui.html, 18.07.2010
2. Brian Marick. *When Should a Test Be Automated?*< http://www.exampler.com>
3. Brian Marick. *A Survey abd Discuss of Automated Testing* <http://www.testingcraft.com/automated-testing-survey.html, 18.07.2010
4. Dalle Olivier & Mrabet Cyrine. "An Instrumentation Framework for Component-Based-Simulations-Based", MASCOTTE project-team, INRIA 2004, France
5. Android Platform Development Kit, "Instrumentation Framework". Google 2008. http://www.netmite.com/android/mydroid/development/pdk/docs/instrumentation_framework.html, 18.07.2010
6. Android Developers, Reference, "Android Test" http://developer.android.com/reference/android/test/package-summary.html, 18.07.2010
7. Autoandroid. *Positron Framework.* http://code.google.com/p/autoandroid/wiki/Positron
8. Selenium. *Selenium-IDE*. http://seleniumhq.org/docs/03_selenium_ide.html, 18.07.2010
9. Android Developers, Dev Guide, "Testing and Instrumentation" http://developer.android.com/guide/topics/testing/testing_android.html, 18.07.2010
10. Research Project: Mobile Paladin. CTI Project 10829.1;3 PFES-ES, 1.12.2009-1.12.2010.

Automating Inspection of Design Models Guided by Test Cases

Anne Rocha[1], Patrícia Machado[1] and Franklin Ramalho[1],

[1] Software Practices Group, Federal University of Campina Grande, Brazil
{annec, patricia, franklin}@dsc.ufcg.edu.br

Abstract. An automated solution to the guided inspection technique is presented in this paper based on the use of Model Driven Architecture (MDA) concepts and technology. Guided inspections use system test cases generated from requirements to conduct inspection of design models. The main expected benefit is to address limitations of model driven testing techniques by supporting: 1) validation of design models construction and behavior (from which test cases are generated) against requirements; and 2) validation of the system test cases and the requirements as design evolves.

Keywords: Guided inspection, design models, test case, MDA, automation.

1 Introduction

Test cases can be generated from design models by using Model-Driven Testing (MDT) [1]. To support the MDT process, there is the Model-Driven Architecture (MDA) framework [2], which performs abstract model transformations to generate concrete models. In MDT, testing models are designed using the Unified Modeling Language 2 (UML2) [3], and they can be transformed to generate test cases.

However, it is not usually possible to perform transformations automatically among artifacts of highest abstraction level, for example, between natural language requirements specification and design models. Therefore, as design models are manually constructed, there is no guarantee that they will reflect the system requirements in a precise and complete manner.

There are several techniques, usually manual, to perform inspection between design UML models and the requirements specification, such as: Object-Oriented Reading Technique (OORT) [4], Perspective-Based Reading (PBR) [5] and Guided Inspection [6]. However, manual techniques have some drawbacks: 1) the inspector needs to be an expert on inspection; 2) some defects cannot be found because of the inspector lack of attention; 3) sometimes, the checklist may not be complete with respect to the system requirements, causing rework when the artifacts are updated. Moreover, techniques based on pre-defined check list can be too constrained.

In this paper, we propose automating the Guided Inspection technique to perform inspections between the requirements specification and the design models in order to check the conformity between them. Since guided inspection is based on test cases,

Short Papers of the 22nd IFIP ICTSS, Alexandre Petrenko, Adenilso Simao, Jose Carlos Maldonado (eds.), Nov. 08-10, 2010, Natal, Brazil.

the inspection can be tailored for each application. Also, the test cases can be re-used as system test cases that are validated with regards to requirements and models.

The paper is structured as follows. Section 2 briefly describes the manual guided inspection technique. Section 3 introduces the automated guided inspection technique. Section 4 discusses related work. Section 5 presents a preliminary evaluation of the proposed technique. Finally, Section 6 presents concluding remarks.

2 Guided Inspection Technique

Inspection techniques are usually applied from a checklist composed of instructions and rules to help the inspector to perform the inspection [7]. These checklist rules, however, often do not represent every behavior of the system. Therefore, the Guided Inspection technique has a particular manner of doing the inspection by analyzing whether the system behavior – which is present in the requirements specification – is also in other software artifacts [8]. The technique uses test cases to guide the inspector. The guiding test cases are 'executed' mentally on the UML diagrams to verify whether it is possible to find each step of the test case on the models.

Fig. 1 shows the phases of the manual Guided Inspection process. In the first phase - the *analysis* - the inspector classifies each use case with regard to its criticality to the system. In the *building* phase, the inspector must write the abstract test cases described in natural language and the analyst has to create the design models to be inspected. In the *execution* phase, the inspector uses the test cases and tries to execute them mentally on the design models. In the *evaluation* phase, the inspector decides whether the models are in conformity with the test case or not.

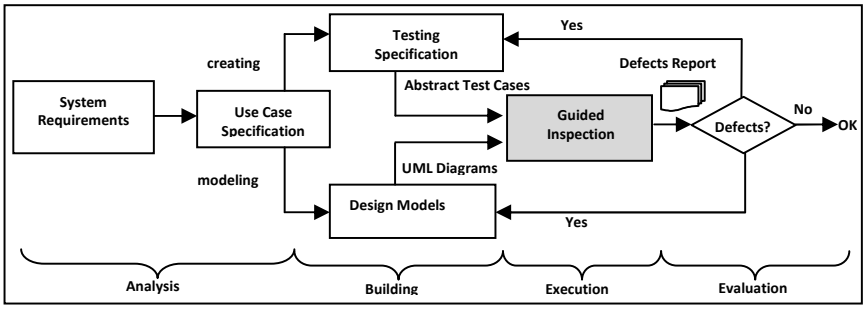


Fig. 1. The four phases of the Guided Inspection process.

3 Automating Guided Inspection

The goal is to automate the 'execution' phase of the manual Guided Inspection technique presented in the previous section. The automation uses the same input and output artifacts from the manual technique. Therefore, the inputs are the test cases

described in natural language and the design models – class diagrams and sequence diagrams – to be inspected. Both artifacts should be created from the requirements specification. Sequence diagrams were chosen, because they can represent the system behavior, while the class diagram is just a structural system representation. The output is a defects report with every inconsistency between the test cases and the design models. The defects found by the automated guided inspection technique can be both behavioral and syntactic ones. The manual technique may only detect behavioral defects. As an example of a behavior defect, a particular action may be present in the use case, but not in the sequence diagram.

Initially, we need to transform natural language test cases into executable ones in order to automatically execute them on the design models inspected. One way to turn a test case into an executable script is by identifying the action semantic of each step. This is made possible by using the Action Semantics from UML2 [3] that describe every available action into a system, for example, the action of creating/destroying an object, to set/read a value, etc. So, each step of the test case is annotated manually with an action semantic by using a custom script language. This is dependent on the class diagrams, since each action is associated with an object or operation. As the inspector annotates the test cases with action semantics, he may not find some object or operation that fully describes the test case written in natural language. This implies the class diagram is not complete and therefore, needs to be fixed.

The next step towards the automation is to run the test case scripts. In literature, several tools have been proposed to simulate and validate design models related to integrity constraints. In order to simulate these models, we need a script containing actions such as creating objects, setting and reading object values, making association between classes or calling an operation. Thus, the test case script created from the annotated action semantics is used. While the script is running, that tool verifies whether some constraint (present in the class diagram) is met or not. The USE (UML-based Specification Environment) Tool [9] was chosen in this work, because it generates automatically a sequence diagram which represents that test case script. This way, the test cases initially described in natural language are now in the same language of the sequence diagrams to be inspected, though at requirements level. This is key to make automatic inspection possible. It is important to remark that the sequence diagram of the test case is not a design diagram since it does not contain design details. When a test case that cannot be simulated, for instance, if class diagram constraints are not met when an operation is called, this may indicate a defect either in the test case (invalid) or in the class diagram. In this sense, all test cases that can be successfully simulated are considered to be valid w.r.t. the structural design.

The Guided Inspection automation is performed using the MDA philosophy, that is, each design sequence diagram is compared (inspection) to the related test case sequence diagram – generated from USE tool – in order to find divergences (defects or alerts) between them. A divergence is transformed into a defect. This is performed by using the ATL (Atlas Transformation Language) language [10] (Fig. 2).

In order to describe the defects found through the inspection, we built a meta-model for the defects report. This meta-model contains the description for all types of defects and alerts found. Thus, the inspection performed between the sequences diagrams are transformed into a defects report model. This model is transformed automatically into a HTML page for visualization.

The defects report contains some types of defects and alerts. The defects types are: (i) *MsgPositionError*, the sent or received lifeline of the message is different from the lifelines on the test case; (ii) *MsgSyntaxError,* related to syntax errors; (iii) *MsgUnfoundError*, it occurs when at least one message is not found; and (iv) *AbstractMsgError*, when a lifeline of an abstract class is created. The alerts types are: (i) *MsgDiffAlert*, messages found on design model were not found in test case sequence diagram; and (ii) *SequenceDiffAlert*, when a message is found in a different order in the test case.

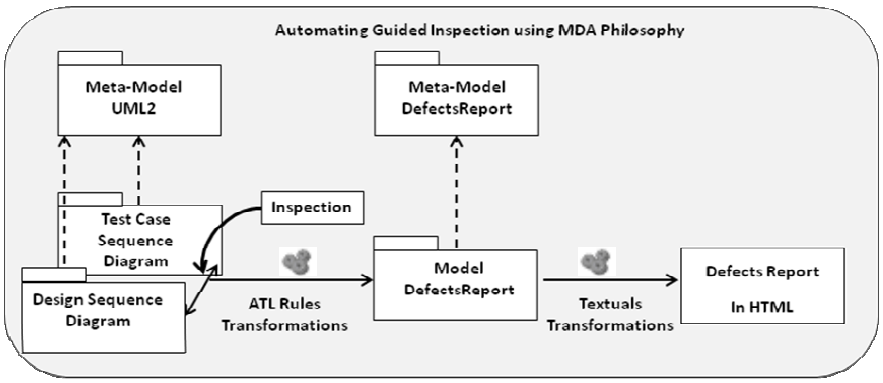


Fig. 2. Automating Guided Inspection using MDA Philosophy.

4 Related Work

This work proposes an automated inspection technique, which inspects UML design diagrams from a requirements specification in a way they can be verified according to syntactic and behaviour aspects based on test cases. Considering these characteristics, the inspection techniques which are more similar to it are Perspective-Based Reading (PBR) and Guided Inspection. Both techniques make manual inspection.

PBR performs inspection from a list of questions on the system related to each stakeholder's perspective (analyst, developer, tester, client etc). These are not specific to any system. Therefore, sometimes these questions are not sufficient for a non-expert inspector to find a large number of defects on the inspected artifacts. However, by using the same questions to any system is an advantage, since it does not require much time from the inspector to start inspecting the system. When a defect must be fixed by an analyst, he may not find, sometimes, this defect in the inspected artifact, since the defects report may not be clear enough. So, the PBR Tool was developed as a support to this problem. It helps to manage each PBR's step and generates a defects report based on the input data. However, this tool does not perform the inspection itself automatically: it is still done manually, as defined in PBR technique.

Guided inspection performs a particular inspection by using test cases to guide the inspector. The main advantage is that these test cases are specific to the system.

However, the inspector needs to mentally verify whether each step of the test case is present in the inspected UML diagram.

5 Experimental Evaluation

To preliminary assess our approach and pinpoint issues to be improved, we have developed case studies to compare the PBR technique to the automated guided inspection (AGI) technique. Below we present the results of a case study set up using a software system with 18 use cases. Two inspectors, one experienced in PBR and another in AGI, conducted the inspection. The main questions investigated are: 1) How long does it take by one inspector to apply each technique, i.e., which technique is more time-consuming? 2) How many defects can be uncovered by each technique?; 3) Which technique presents the best performance in terms of the rate of defects per time spent? To answer these questions, the verified metrics were: total time, number of defects and performance. Table 1 presents the results of this evaluation.

Table 1. Evaluation results regarding the AGI and PBR Techniques.

Metrics	**PBR**	**AGI**
Total Time (T)	0,26+1,67 = 1,93 h	4,96+0,15 = 5,11 h
Number of Defects (N)	14	35
Performance (N/T)	7,25	6,85

The total time is the sum of the preparing time and the inspecting time. PBR's preparing time is considerably smaller than the AGI's, because in the former the inspection artifacts are the same independently of the system, whereas in AGI, it is necessary to perform manual annotation with the action semantics of the test cases. Thus, AGI's preparing time takes longer than PBR's. However, PBR's inspection time takes longer than AGI's, because the latter inspection execution was automated. AGI's inspection time takes approximately 30 seconds for each system use case.

As for number of defects, note that by using AGI it was possible to find more than twice as many defects as PBR. The performance is the relation between the number of defects and the total time. Although PBR takes less time (1,93h vs 5,11h), the performance of the techniques was similar (7,25 vs 6,85), as AGI found more defects.

Even though no definite conclusions can be reached from this case study, the results point out that AGI may be a good choice when uncovering more defects is important to the project as far as resources are available and the costs can be compensated by avoiding escaped defects. On the other hand, PBR is more adequate for quick inspections since it seems to be less accurate than AGI with less effort.

6 Concluding Remarks

This paper presents an approach to automate and extend the guided inspection technique to perform semantic and syntax inspection between design UML diagrams

and requirement specification. Automation is based on MDA concepts and technology focusing on the use of Action Semantics from UML2 and model simulation through the USE tool. Since the manual guided inspection uses abstract test cases to guide the inspector, we use Action Semantics to annotate the semantic of each test case step, turning those test cases into executable scripts. The test cases are transformed into sequence diagrams by using the USE tool. It allowed those abstract test cases to be ready for inspection, because the UML design diagrams to be inspected are also sequence diagrams. Since sequence diagrams represent the system behavior, it is possible to identify conformity defects on these artifacts. The automated guided inspection execution is made by using the MDA philosophy.

A preliminary experimental evaluation was performed by comparing AGI to PBR. Both techniques have positive and negative points, because while PBR was performed faster than AGI, the latter technique found more than twice as many defects as PBR.

As further work, it is necessary to improve the phase of annotating the test cases with action semantics in order to make it less time-consuming. The use of DSLs can make automation of this phase possible, reducing considerably the costs of AGI. Additionally, experimental evaluation using real-world case studies are needed to assess the benefits that will make it clear how the technique can be used in practice.

Acknowledgments. This work was partially supported by the National Institute of Science and Technology for Software Engineering (INES[1]), funded by CNPq, grant 573964/2008-4.

References

1. Baker, P., Dai, Z. R., Grabowski, J., Haugen, Ø., Schieferdecker, I., Williams, C.: Model-Driven Testing: Using the UML Testing Profile. Springer, New York (2008)
2. Kleppe, A., Warmer, J., Bast, W.: MDA Explained: The Model Driven Architecture: Practice and Promise. Addison Wesley, Boston (2003)
3. Object Management Group, UML superstructure, v2.1.1., Technical report, http://www.omg.org/cgi-bin/doc?formal/07-02-05
4. Travassos, G. H., Shull, F., Carver, J., Basili, V. R.: Reading techniques for OO design inspections. In: 14th Brazilian Symposium on Software Engineering, Rio de Janeiro (2002)
5. Costa, P., Shull, F., Melo, W.: Getting requirements right: The Perspective-Based Reading technique and the rational unified process. Technical report, IBM (2006)
6. McGregor, J. D., Sykes, D. A.: A Practical Guide to Testing Object-Oriented Software. Addison-Wesley, New York (2001)
7. Sauer, C., Jeffery, D. R., Land, L., Yetton, P.: The effectiveness of software development technical reviews: A behaviorally motivated program of research. In: IEEE Transactions on Software Engineering, IEEE Press, New York (2000)
8. Major, M. L., McGregor, J. D.: Using guided inspection to validate UML models. In: 25th Annual Software Engineering Workshop, pp. 485--507. Greenbelt (1999)
9. Gogolla, M., Büttner, F., Richters, M.: USE: A uml-based specification environment for validating UML and OCL. Sci. of Comp. Programming, pp. 69(1-3):27-34. (2007)
10. AMMA Project, Atlas Transformation Language. http://www.sciences.univnantes.fr/lina/atl/

[1] http://www.ines.org.br/

Concurrent Software Testing: A Systematic Review*

Maria Brito, Katia Felizardo,
Paulo Souza, and Simone Souza

Instituto de Ciências Matemáticas e de Computação
Universidade de São Paulo – (ICMC/USP)
Caixa Postal 668 – 13560-970 – São Carlos – SP – Brazil
{masbrit,katiarf,pssouza,srocio}@icmc.usp.br
http://www.icmc.usp.br

Abstract. Software testing applied for concurrent programs is a challenging activity. Considering the relevance of concurrent programs testing, several research have been conducted in this area, especially involving adaptation of the techniques and criteria applied in sequential programs. This paper presents a systematic review, which is a technique coming from Evidence-Based Software Engineering, aiming to find out research about testing in this context. As result, the review identified testing criteria, bugs taxonomy and testing tools that can provide the foundation for new research in the area of concurrent program testing, such as definition of new criteria and new testing tools.

Keywords: software testing, concurrent program, systematic review

1 Introduction

The objective of the concurrent programming is to increase application performance, allowing better use of available resources. A concurrent application consists of several processes that interact to solve a problem, reducing the computational time in several application domains, such as: web services, corporative systems, systems based on service oriented architecture (SOA) and complex systems, such as weather forecast, dynamic molecular simulation, bio-informatics and image processing. Moreover, the current demand for distributed applications in domains as web services and corporative systems has increased the interest for concurrent programming.

There are two paradigms for interaction among concurrent processes: shared memory and message passing. When message passing is considered, the communication (and synchronization) among concurrent processes occurs by exchanging messages. In the case of shared memory, the communication occurs by sharing address spacing and the shyncronization occurs by primitives, such as semaphores or monitors. These differents aspects need to be considered for the

* The authors are financially supported by FAPESP and CNPq – Brazil.

proposition of the testing mechanisms for concurrent programs, and features, such as nondeterminism, synchronization and communication. In this direction, some work prior to 2000 have defined a bugs taxonomy, criteria and testing tools for concurrent programs [21, 9, 3, 22]. However, it is important to know the current research about the area.

This paper contributes in this direction, providing a systematic review about research related to testing in context of concurrent programs, considering research related to testing criteria, bugs taxonomy and testing tools. According to our knowledge this is the first effort to provide a Systematic Review in this area. Our objective is to analyse the existing approaches to testing in concurrent software, discussing the significant issues related to this area and providing evidence that can serve as a basis for innovative research activities.

2 Systematic Review: Planning and Conducting

A systematic review is a process of assessment and interpretation of all available studies related to a research question or subject of specific interest, providing a background for further investigation [8]. This review intends to identify current studies related to testing criteria, bugs taxonomy and testing tools for concurrent programs. The objective is to facilitate the development of new research in this area, from of the relationship among proposed approach. This review was performed according to the process defined in [8]. For reasons of space, we will present a summary of the review conducted, for details see [2].

Three research questions were formulated to the objectives of the systematic review: 1) What testing criteria were proposed to test concurrent programs? 2) What type of bugs taxonomy related to concurrent programs were identified? 3) What tools were proposed for concurrent program test? Based on the research questions, some keywords were extracted and used to search the primary studies. Primary studies (in the context of evidence) is an empirical study investigating a specific research question [8]. The initial set of keywords was: *"concurrent program", "bug taxonomy" and "test"*. During the preliminary search conducted other keywords were added to improve the search string general. The digital libraries considered in this review were: ACM Digital Library (portal.acm.org), IEEE eXplore (ieeexplore.ieee.org), Engineering Village (engineeringvillage.com) and SCOPUS (scopus.com). After of the definition of search string, the searches were performed. All the results of the search process were documented, therefore, others research can to acess this documentation and, if necessary, to repeat the review. The selection process of studies was validated using VTM technique [6] that help in new review of the studies. After this phase, 14 primary studies were selected for data extraction.

3 Software Testing for Concurrent Programs

This section summarises the information extracted of the primary studies. The Table 1 presents information about the studies selected organized by: category

(related to research question), paradigm of the parallel program, testing technique used, language considered, test model defined, funcionality of approach and how the validation of approach is shown. In the following, the studies identified are described.

Table 1. Primary Studies Summary

Paper	Category	Paradigm	Technique	Language	Test Model	Funcionality	Validation
[10]	Testing criteria	Message passing	Structural		Synchronization graph	Validates SYN-sequence events	Examples
[12]	Testing criteria	Shared memory	Structural		Concurrent faults	Exercises synchronization sequence of the concurrent program	Figure illustrates
[23]	Testing criteria	Shared memory	Structural		PPFG (parallel graph)	Exercises pairs definition-use.	Case study
[20]	Testing criteria	Shared memory	Structural		CMFG (Concurrent Module Flow Graph)	Exercises all paths of the concurrent program	Case study
[17]	Testing criteria	Shared memory	Structural		PCFG_SM (parallel graph)	Exercises features of the shemaphore based programs	Case study
[18]	Testing criteria		Model based		Statechart	Exercises synchronization sequence	Case study
[15]	Testing criteria		Model based		Timed interface automaton and CTL	Exercises pair and paths of the send/receive primiteves between process	Case study
[16]	Testing criteria		Model based		Timed interface automaton and CTL	Extends [15] to consider timed interface automaton	Practical study
[1]	Testing criteria and Taxonomy bug	Shared memory	Error based	Java		Mutation operators exercise concurrent aspects and the bugs not availabre in [5]	Case study
[11]	Taxonomy bug	STM		C		Related to interleaving sequences in STM and problems to data update	Practical study
[13]	Taxonomy bug			C/C++		Present deadlock bugs and non-deadlock concurrency bugs	practical study
[19]	Testing tool	Shared memory	Structural	Java		Calls are inserted the scheduling function in places of program, that uses generation of pseudo-random numbers	Case study
[7]	Testing tool	Shared memory	Structural	Java		Provides a framework to implement different dynamic predictive analysis to obtain a set of statements of the program involved in a potential error	Case study
[14]	Testing tool	Shared memory	Structural	Java		Conducts analysis of trace for a list of interleavings and then explores these interleavings using controlled test	Case study

3.1 Testing Criteria

Testing criterion defined by Ling et al. [10] considers timing-sequences, Lu et al. [12] explors also test sequences adding the concept of representative sequences, and considers differents features of interleavings in an abstraction hight level.

Representative test sequence is a subset of all possible interleaving of the concurrent events that define the concurrent program behavior. Yang and Pollock [23] present the criterion all-uses for concurrent programs, in more detail that [12]. The criterion all-uses is more explored in [17] considering differents types of associations between definition and use of variables. Takahashi et al. [20] propose a new approach that uses the concept of abstract block, considering hardware and low level of commands in order to reduce the number of combinations between statements of the concurrent program that must be tested.

In context of model based testing Seo et al. [18] use statecharts and propose an approach that uses automata limiting the state-explosion problem. Robinson-Mallett et al. [15] focus on the comunication testing between components selecting representative test sequence from open interface automata with high efectiveness. The advantage of this approach [15] is the generation of a potentially minimal set of test cases. Extending this work, Robinson-Mallett et al. [16] takes advantage of the testability of communication between components for the integration testing. About error based testing, Bradbury [1] proposes a mutation operator set based on concurrent faults related to concurrent aspects, such as: concurrent method, calls of the concurrent method and critical region.

3.2 Bug Taxonomy

Bradbury [1] used a bug pattern taxonomy defined in Farchi et al. [5] to analyse the IBM Concurrency Benchmark [4] and to define bugs in context of *Java* adicioning aspects not available when the taxonomy was proposed, such as other signals notify that can are missing, problems with starvation or number of resources for threads, or incorrect count initialization. Lourenço and Cunha [11] present bugs set based on Software Transactional Memory (STM). STM is a control mechanism similar to competitive transactions of databases, but in this case to control access to shared memory in concurrent computing. Lu et al. [13] present a study of the concurrent bugs in real application, considering another aspects not explored in [5], such as threads number, variables number and acess involved in this bugs.

3.3 Testing Tools

Stoller [19] presents the *rstest* tool that similary the tool *CalFuzzer* [7] uses scheduling. *CalFuzzer* works with the concept of active testing: first uses predictive or dynamic program analyses to identify potential concurrency bugs and after uses the reports from these analyses to explicitly control the underlying scheduler of the concurrent program to discover real concurrent bugs. The disadvantage of *CalFuzzer* is that it control execution by scheduling one thread at a time, leaving out the advantages of multicore machine in testing. Park et al. [14] proposes the *CTrigger* that similarly the *CalFuzzer* works in two phases: trace analysis to obtain a list of unserializable interleavings and analysis of these unserializable interleavings using controlled testing. *CTrigger* controls execution

by suspendig the execution of the thread in determined places, increasing the probability of the occurrence of the targeting unserializable interleaving.

4 Conclusion

This paper presented a systematic review aiming to find out research related to testing criteria, bugs taxonomy and testing tools for concurrent software testing, providing foundation for new research in this area. Basically, testing criteria found are related to structural, model based and error based testing techniques. These criteria explore comunication inter process, components of the concurrent programs and message passing, in most cases, considering shared memory programs. It was observed that 64.3% of the studies present testing criteria. In this studies, 55.5% are structural testing criteria, 33.3% are model based testing criteria and 11.1% present error based testing criteria. Considering bugs taxonomy, 21.4% of the studies present definition of the typical bugs for concurrent programs. Of the studies 21.4% present testing tool.

From results obtained in this review, it is observed a lack of the testing criteria and tools for concurrent programs, considering intrinsic aspects of these programs. We plan realize other review more complete, considering in the research string all aspects of concurrent programs. Also, there is a lack of experimental studies to evaluate testing criteria and bugs taxonomy. Experimental studies are fundamental to provide information about application cost, efficacy and complementary aspect of the testing criteria. Thus, we plan the development of an experimental study, using bugs taxonomy of the primary studies, to evaluate of the efficacy in fault reveal of the testing criteria for concurrent programs. We hope this study will contribute to the definition and the evolution of testing strategies in context of concurrent programs.

References

1. Bradbury, J. S. Using program mutation for the empirical assessment of fault detection techniques: a comparison of concurrency testing and model checking. PhD thesis, Kingston, Ont., Canada (2007)
2. Brito, M. A. S., Felizardo, K. R., Souza, P. S. L., and Souza, S. R. S.: Concurrent Software Testing: A Systematic Review. Technical report, University of São Paulo, Institute of Mathematical Sciences and Computing - ICMC/USP. Brazil (2010)
3. Chung, C-M. and Shih, T. K. and Wang, Y-H. and Lin, W-C. and Kou, Y-F.: Task Decomposition Testing and Metrics for Concurrent Programs. In: Fifth ISSRE, pp. 122–130 (1996)
4. Eytani, Y. and Ur, S.: Compiling a benchmark of documented multi-threaded errors. In: Parallel and Distributed Processing Symposium, International, 17, 266 (2004)
5. Farchi, E., Nir, Y., and Ur, S.: Concurrent errors patterns and how to test them. In: Proceedings of the 17th International Symposium on Parallel and Distributed Processing, Washington, DC, USA (2003)
6. Felizardo, K. R. and Maldonado, J. C.: A visual approach to assist the selection review of primary studies in systematic reviews. In: VI Experimental Software Engineering Latin American Workshop (ESELAW'09), pp. 83–92 (2009)

7. Joshi, P., Naik, M., Park, C.-S., and Sen, K.: Calfuzzer: An extensible active testing framework for concurrent programs. In: Proceedings of the 21st International Conference on Computer Aided Verification. pp. 675–681, Berlin, Heidelberg (2009)
8. Kitchenham, B. and Charters, S.: Guidelines for performing systematic literature reviews in software engineering. Technical Report EBSE 2007-001, Keele University and Durham University Joint Report (2007)
9. Krawczyk, H. and Wiszniewski, B.: Classification of Software Defects in Parallel Programs. Technical report, Faculty of Electronics, Technical University of Gdansk, Poland, 2 (1994)
10. Liang, Y., Li, S., Zhang, H., and Han, C.: Timing-sequence testing of parallel programs. Journal of Computer Science and Technology. 15, 1, 84 – 95 (2000)
11. Lourenço, J. and Cunha, G.: Testing patterns for software transactional memory engines. In: Proceedings of the 2007 ACM workshop on Parallel and distributed systems: testing and debugging. pp. 36–42, New York, USA (2007)
12. Lu, S., Jiang, W., and Zhou, Y.: A study of interleaving coverage criteria. In: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering. pp. 533–536, New York, USA (2007)
13. Lu, S., Park, S., Seo, E., and Zhou, Y.: Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems, pp. 329–339, New York, NY, USA (2008)
14. Park, S., Lu, S., and Zhou, Y.: Ctrigger: exposing atomicity violation bugs from their hiding places. In: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems, pp. 25–36, New York, NY, USA (2009)
15. Robinson-Mallett, C., Hierons, R. M., and Liggesmeyer, P.: Achieving communication coverage in testing. SIGSOFT Softw. Eng. Notes. 31, 6, 1–10 (2006)
16. Robinson-Mallett, C., Hierons, R. M., Poore, J., and Liggesmeyer, P.: Using communication coverage criteria and partial model generation to assist software integration testing. Software Quality Control. 16, 2, 185–211 (2008)
17. Sarmanho, F. S., Souza, P. S., Souza, S. R., and Simão, A. S.: Structural testing for semaphore-based multithread programs. In: Proceedings of the 8th international conference on Computational Science, Lecture Notes in Computer Science, pp. 337–346. Springer-Verlag, Krakow (2008)
18. Seo, H.-S., Chung, I. S., and Kwon, Y. R.: Generating test sequences from statecharts for concurrent program testing. IEICE - Trans. Inf. Syst. E89-D, 4, 1459–1469 (2006)
19. Stoller, S.: Testing concurrent java programs using randomized scheduling. Electronic Notes in Theoretical Computer Science. 70, 4, 149–164 (2002)
20. Takahashi, J., Kojima, H., and Furukawa, Z.: Coverage based testing for concurrent software. In: Proceedings of the 2008 The 28th International Conference on Distributed Computing Systems Workshops, pp. 533–538, Washington, USA (2008)
21. Yang, R-D. and Chung, C-G.: Path Analysis Testing of Concurrent Programs. Information and Software Technology. 34, 1, 43–56 (1992)
22. Yang, C-S. and Souter, A. L. and Pollock, L. L.: All-Du-Path Coverage for Parallel Programs. In: Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis, pp. 153–162. ACM-Software Engineering Notes, New York, NY, USA (1998)
23. Yang, C.-S. and Pollock, L. L.: All-uses testing of shared memory parallel programs. Software Testing Verification and Reliability. 13, 1, 3–24 (2003)

Evolving a Computerized Infrastructure to support the Selection of Model-Based Testing Techniques

Arilo Claudio Dias-Neto[1], Guilherme Horta Travassos[2]

[1] Computer Science Department – Federal University of Amazonas (DCC/UFAM)
Campus Universitario, 69077-000 – Manaus –AM – Brazil.
arilo@dcc.ufam.edu.br

[2] Federal University of Rio de Janeiro – PESC/COPPE/UFRJ
P.O. Box 68511 – 21.941-972 – Rio de Janeiro – RJ – Brazil.
ght@cos.ufrj.br

Abstract. This paper describes a computerized infrastructure to support the combined selection of Model-based Testing (MBT) Techniques to be jointly used for a software project. The decision making can be supported by a selection approach providing the generation of indicators and charts which MBT techniques could be considered regarding the software project's characteristics and requirements. Results of an observational study performed with software engineers suggest the proposed infrastructure could be able to guide the selection of the best-suited combination of MBT techniques for a software project, can reduce about 50% of the selection time in comparison with manual selection, and can approximate the time spent by software engineers with different experience backgrounds while performing the MBT selection task.

Keywords: Software Technologies Selection, Software Testing, Model-based Testing, Experimental Software Engineering.

1 Introduction

The decision regarding which testing techniques to adopt in a software project represents an important factor that may directly influence software process effectiveness and software product quality [8]. Testing activities usually require the combined use of techniques with the purpose of increasing the testing coverage and quality [6]. However, the combination of testing techniques may increase testing effort and cost when the requirements to use these techniques are not explicit in the development software process. Therefore, the selection of testing techniques requires a carefully analysis based on the software project's characteristics/needs [5].

Testing techniques may be classified into different subcategories accordingly specific characteristics that may influence on their selection for a software project (e.g.: requirements-based, code-based, and model-based). This work is focused in the Model-based Testing (MBT) subcategory, that represents a type of testing strategy in which test cases are derived in whole or in part from a model describing some aspect (e.g.: functionality, security, performance, etc.) of a software product [7]. Contextualizing the software technologies selection's scenario for MBT, additional challenges for the selection of MBT techniques for a software project can be observed because the specific characteristics of this testing technique subcategory.

Based on these additional challenges, Dias-Neto and Travassos have developed an approach called *Porantim* [2;3] to support the combined selection of MBT techniques for software projects. *Porantim* promotes the interrelating the characteristics of a software project and MBT techniques and the calculus of indicators suggesting the proximity between a software project and MBT techniques characteristics. Doing so, it may support the decision making regarding which MBT techniques would be best

Short Papers of the 22nd IFIP ICTSS, Alexandre Petrenko, Adenilso Simao, Jose Carlos Maldonado (eds.), Nov. 08-10, 2010, Natal, Brazil.

suited for a software project. However, the manual selection of MBT techniques for a software project using *Porantim* is an error-prone task. This scenario has motivated the development of a computerized infrastructure to implement the *Porantim* approach, aiming at improving the MBT techniques selection process and minimizing errors during *Porantim*'s application.

This paper describes the facilities implemented in a computerized infrastructure (Maraká [1]) developed to support *Porantim*, including an observational study conducted with software engineers from an international software organization that uses MBT techniques. The results suggest the proposed infrastructure can represent a feasible mechanism to support the selection of MBT techniques for a software project.

Besides this Introduction, this paper is organized as follows: Section 2 summarizes *Porantim*, an approach to support the selection of MBT techniques for software projects. Section 3 describes the computerized infrastructure (Maraká) that has been evolved to support the *Porantim* functionalities. Section 4 describes a study performed with experts in MBT from a software organization to evaluate the computerized infrastructure. Finally, Section 5 presents the conclusions and future works.

2 *Porantim*: Approach to Support the MBT Techniques Selection

The approach *Porantim*[1] [2;3] aims at supporting the selection of MBT techniques for software projects. It represents an evolution of the characterization schema of testing techniques proposed by Vegas and Basili [8]. *Porantim* is based on two elements (Fig. 1): (1) a body of knowledge on MBT techniques extending the original one proposed in [8] working as a repository of techniques that may be used in software projects and a (2) process to support the selection of MBT techniques (not provided in the original approach proposed in [8]).

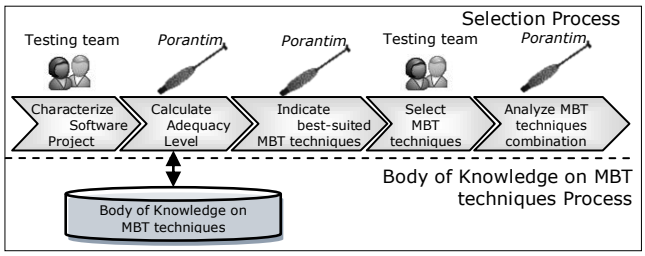


Fig. 1. *Porantim*'s Elements [2]

The first element, *Body of Knowledge*, consists in a repository of MBT techniques. Each technique is described as a set of characterization attributes weighted according to their importance to support the selection task. The definition of the MBT techniques characterization attributes and their weights were obtained through a survey performed with experts in the MBT field. The current content of this body of knowledge consists in 219 characterized MBT techniques identified by a systematic literature review (secondary study). Detailed information about the body of knowledge on MBT techniques can be found in [3].

The second element, *MBT Techniques Selection Process*, consists in a set of 5 activities (see Fig. 1) promoted by *Porantim* to provide knowledge and guidance in the selection of MBT techniques for software projects. This process is based on the generation of indicators to support the analysis of MBT techniques' adequacy levels

[1] *Porantim* is a piece of wood used by the *Sateré-Mawé* (Amazon Indians) that is used in the *Wat'amã* ritual, a challenge set to select tribe warriors.

and the impact analysis of their combination in the testing process for a software project. Detailed information regarding *Porantim*'s selection process, the available mathematical formulas to calculate the indicators supporting the MBT techniques selection can be found in [2]. *Porantim* was evaluated by an experimental study, and the results suggested it contributes to increase efficiency and effectiveness of the MBT techniques selection process in a software project [4].

The next section presents the computerized infrastructure developed to semi-automate the selection of MBT techniques using *Porantim*.

3 Computerized Infrastructure to support *Porantim*

The semi-automate support to *Porantim* approach was developed as an evolution of Maraká [1], a computerized infrastructure to support software testing planning and control.

Maraká was built based on the open source framework Joomla! (www.joomla.org) that uses the software technologies PHP+MySQL, which characterize the construction platform [1]. In this way, Maraká was evolved to provide facilities to perform the selection of testing (MBT) techniques through the use of *Porantim* approach.

3.1 Configuration of *Porantim*

It consists in the construction and maintenance of MBT techniques' repository. Accessing this option, the MBT techniques available at the Maraká's repository are listed. Software engineers can add, filter, edit, exclude, and import MBT techniques from a specific database built by using the JabRef Tool (http://jabref.sourceforge.net).

Moreover, it is possible to adjust the MBT techniques characterization attributes weights, adapting them according to the characteristics and needs of the software organization using *Porantim*. The sum of all characterization attributes' weight must be 1, representing 100%.

3.2 Supporting the MBT Techniques Selection Process

The selection of testing techniques usually is accomplished in the *Test Planning* activity. In this context, Maraká offers a testing process containing the activity *Plan Tests* and one sub-activity *Define Testing Techniques*. There are two possible options to describe the testing techniques will be used in the software project: to define the testing techniques by manual descriptions or using the *Porantim* selection process, when applied to the context of MBT techniques. This feature implements the "Selection Process" element, presented in Fig. 1, and it is divided into three steps:

STEP 1) Software Project Characterization: testing team can define the tests characteristics and requirements for the software project they want to apply MBT techniques, filling the characterization attributes for the software project as defined in the *Porantim*'s configuration screen (Section 3.1).

STEP 2) Selection of MBT Techniques: the adequacy levels of all MBT techniques filtered from the repository are automatically calculated, after the software project characterization performed in step 1, using the mathematical formulas

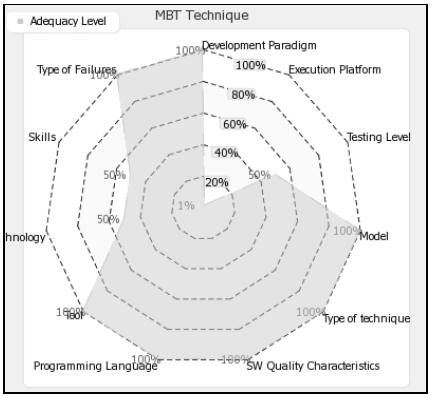


Fig. 2. Spider Chart (MBT Tech. Adequacy)

provided by *Porantim* [2]. After that, the MBT techniques are showed ordered by their adequacy level. For each MBT technique, the computerized infrastructure provides a link to visualize a detailed analysis of the selected MBT technique adequacy accordingly each characterized software project attribute using a graphical representation: Spider Chart (Fig. 2). In this step, the testing team needs to select which MBT techniques will be used in the software project. This step implements the activities 2, 3, and 4 of the "Selection Process" element (Fig. 1). The testing team may use this information to decide about the MBT techniques for the software project.

STEP 3) Combining MBT Techniques: it is performed an impact analysis of the selected MBT techniques calculating 3 indicators: *Testing Requirements Coverage*, *Saved Modeling Effort* and *Testing Team Adequacy*. These indicators represent numbers between 0 and 100%. In order to simplify their representation, the values were divided into four ranges: [100-75%]: High; [75-50%]: Medium; [50-25%]: Low; [25-0%]: Very Low. Moreover, we decided to use a Gauge Chart[2] to graphically represent these indicators. The chart's needle indicates the value obtained in each moment (Fig. 3). Using this information, the testing team can have more information to decide on which MBT techniques could be combined for the software project.

If the testing team confirms the MBT techniques selection, its choices will be stored in the software project test plan, and the infrastructure execution flow is redirected to the next testing process activities monitored by Maraká.

One evaluation version of this extended version of Maraká is available http://ese.cos.ufrj.br/porantim, using the login *guest* and password *porantim*.

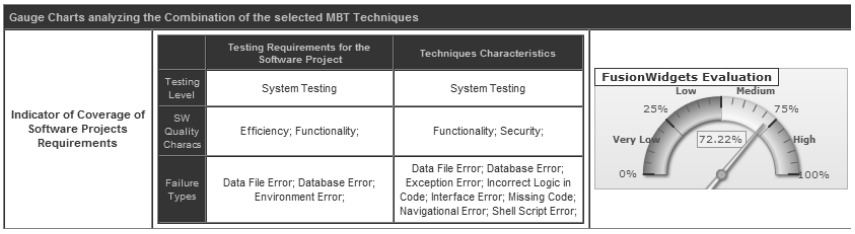


Fig. 3. Example of Analysis of MBT Techniques Combination

4 Evaluation of *Porantim* features in Maraká

To evaluate the evolved version of Maraká containing facilities to support the MBT techniques selection process provided by *Porantim*, we conducted in June of 2009 an observational study in an organization that applies MBT in its software projects located in USA. Its results were important to obtain an initial evaluation of the computerized infrastructure, analyzing if it could be ready to be applied in more complex evaluation studies in the industry.

Two test manager specialists in MBT participated of this study. The study's strategy consisted in selecting two MBT techniques for a software project. Each participant performed two selections, one without using the computerized supporting and another one using the presented infrastructure. Different software projects were used in each trial. After the selection tasks, the selected MBT techniques would be compared to an oracle (Table 1) previously defined by another specialist in MBT.

In the evaluation study, we used two software projects (*Parking* and *Video Management* – also used in the *Porantim*'s experimental study [4]) and five real MBT techniques (a sub set of the MBT techniques available in the Maraká's repository).

[2] We used an evaluation version of *FusionWidgets* component (http://www.fusioncharts.com/widgets/).

Table 1. Evaluation Study's Oracle

SW Projects	MBT Techniques used	Best-suited MBT Techniques
Video	• **Technique 1:** Abdurazik and Offutt (2000), "Using UML Collaboration Diagrams for Static Checking and Test Generation", 3rd UML'2000. • **Technique 2:** Briand and Labiche (2001), "A UML-Based Approach to System Testing", 4th UML'2001. • **Technique 3:** Hartman and Nagin (2004), "The AGEDIS tools for model based testing". SIGSOFT SE Notes. • **Technique 4:** Kim *et al.* (1999), "Test cases generation from UML state diagrams", IEE Proceedings: Software. • **Technique 5:** Chung *et al.* (1999), "Testing of Concurrent Programs Based on Message Sequence Charts", PDSE.	Techniques 1 and 2
Parking	• **Technique 2:** Briand and Labiche (2001), "A UML-Based Approach to System Testing", 4th UML'2001. • **Technique 3:** Hartman and Nagin (2004), "The AGEDIS tools for model based testing". SIGSOFT SE Notes. • **Technique 5:** Chung *et al.* (1999), "Testing of Concurrent Programs Based on Message Sequence Charts", PDSE. • **Technique 6:** Bousquet and Zuanon, N. (1999), "An Overview of Lutess: A Specification-Based Tool for Testing Synchronous Software". ASE. • **Technique 7:** Parissis and Vassey (2003), "Thoroughness of Specification-Based Testing of Synchronous Programs", ISSRE.	Techniques 6 and 7

4.1 Evaluation Study's Data Analysis

Table 2 presents the results provided by each participant by a questionnaire filled in during the study execution.

Table 2. Results obtained in the computerized infrastructure evaluation

Item evaluated	SW Engineer 1	SW Engineer 2
Phase 1 – Manual selection of MBT Techniques		
SW Project used in the MBT techniques manual selection	Video	Parking
Total Time (min) spent for selection process execution	25 minutes	15 minutes
MBT techniques selected	Techniques 1 and 2	Techniques 6 and 7
Phase 2 – Selection of MBT Techniques using the infrastructure		
SW Project used in the MBT techniques selection using Maraká	Parking	Video
Total Time (min) spent for selection process execution	14 minutes	7 minutes
MBT techniques selected	Techniques 6 and 7	Techniques 1 and 2

Although the small number of participants, the collected data indicate the use of the infrastructure could reduce about half the total time to select MBT techniques when compared to the manual selection (Fig. 4).

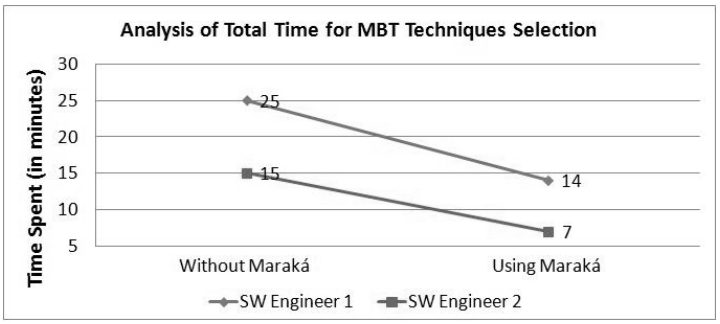


Fig. 4. Analysis of MBT Techniques Selection Time

The data analysis considering the participants' experience level indicates the use of the infrastructure by the SW Engineer 1 (test manager who participated of less software projects using MBT) makes the total time to perform the MBT techniques selection similar to the time obtained without the use of the infrastructure by the SW

Engineer 2 (test manager who participated of more software projects using MBT) for the software project *Parking*. On the other hand, when the SW Engineer 2 used the infrastructure to support the selection of MBT techniques, the total time reduced significantly (around 3.5 times lower).

5 Conclusions

This paper presented the evolution of the computerized infrastructure Maraká by implementing a repository of MBT techniques and a component to apply the MBT techniques selection approach *Porantim*.

Maraká's facilities supporting *Porantim* are concerned with mechanisms to allow:

- The automated calculus of adequacy levels between a software project previously characterized and MBT techniques available in a repository, besides the visual representation of these adequacy levels using spider charts.
- The automated calculus of indicators to support the analysis of combining MBT techniques for a software project, and the visual representation of these indicators using Gauge charts.

We cannot claim the results obtained in the observational study are conclusive or can be generalized, because we had few data points and they must be contextualized to a scenario organized by the researchers. However, they were important by allowing performing an initial computerized infrastructure evaluation. Thus, they allow us to plan future experimental studies to evaluate new aspects of the proposed infrastructure in environments similar to real software development projects.

The next steps regarding this research concern with the evaluation of Maraká (with *Porantim*) into real software projects and different testing team profiles, its extension providing the selection of MBT techniques for software projects portfolio, and the optimization of the selection process in Maraká using strategies of the Combinatorial Analysis field, such as local search heuristics.

Acknowledgments. Our thanks to CNPq (Grant 75459/2007-5), FAPERJ, FAPEAM, and SCR for the financial support. We would like to thank Lucas Paes and Priscila Pecchio for their contribution for the development of the computerized infrastructure.

References

1. Dias-Neto, A. C.; Travassos, G. H. (2006), "Maraká: A Computerized Infrastructure to support Software Testing Planning and Control". In: Brazilian Symposium on Software Quality, Vila Velha (in Portuguese).
2. Dias-Neto, A.; Travassos, G. (2009), "*Porantim*: An Approach to Support the Combination and Selection of Model-Based Testing Techniques", In: 4th Workshop on the Automation of Software Test, v. 04, Vancouver, May.
3. Dias-Neto, A.C.; Travassos, G.H. (2009), "Model-based Testing Approaches Selection for Software Projects", In: Information and Software Technology, v. 51, pp. 1487-1504.
4. Dias-Neto, A.C.; Travassos, G.H. (2009), "Evaluation of {model-based} Testing Techniques Selection Approaches: an External Replication", In: International Symposium on Empirical Software Engineering and Measurement, Lake Buena Vista, EUA, October.
5. Menzies, T., Owen, D., Cukic, B. (2002), "Saturation Effects in Testing of Formal Models". In: 13th international Symposium on Software Reliability Engineering (ISSRE'02), Washington, DC, p. 15.
6. Myers, G.J.; Sandler, C.; Badgett, T; Thomas, T.M. (2004), "The Art of Software Testing". John Wiley & Sons, Inc., New Jersey.
7. Utting, M.; Legeard, B.; (2007), "Practical Model-Based Testing: A Tools Approach", ISBN-13: 978-0-12-372501-1, Morgan-Kaufmann.
8. Vegas, S.; Basili, V. (2005), "A Characterization Schema for Software Testing Techniques", Empirical Software Engineering, v.10 n.4, p.437-466, October.

Using Probabilistic Model Checking for Safety Analysis of Complex Airborne Electronic Systems

Fabiano Costa Carvalho[1] and José M. Parente de Oliveira[1]

[1]Instituto Tecnológico de Aeronáutica
Praça Marechal Eduardo Gomes, 50 - Vila das Acácias
São José dos Campos, SP – Brasil
fabiano@mectron.com.br, parente@ita.br

Abstract. The increasing complexity of airborne electronic systems pushes the need for advanced techniques to address reliability and safety issues before physical implementation and deployment. In this context, this paper proposes a systematic approach based on probabilistic model checking for creating and analyzing quantitative fault models at early phases of the design lifecycle. Moreover, this paper provides a case study with the intention to exercise the proposed approach and to evaluate its adequacy to the industrial context.

Keywords: Probabilistic Model Checking; Safety Management; Airborne Systems Design

1 Introduction

Airborne electronic systems shall be designed to perform safety-critical functions while ensuring that the risk of fatal accidents is minimized. As design complexity increases, efficiency of traditional methods for safety assessment is limited since they strongly rely on human skills. Recent work addresses this issue by combining model-based development paradigm with formal methods. In [1] authors focus on integration of design and safety assessment processes in a common framework for creating and managing fault models that are suitable for model checking. Similarly, [2] proposes a model-based approach in which fault models are created directly from design models elaborated in Simulink or SCADE, and then translated to input models for NuSMV model checker and PVS theorem prover.

In this paper we consider a model-driven architectural safety analysis approach based on PRISM [3], a probabilistic model checker that is suitable for representing and analyzing systems that exhibits probabilistic behavior. We claim that our approach differs from related work by providing quantitative analysis features.

This paper is organized as follows: Section 2 briefly introduces the PRISM language. Section 3 describes the proposed approach of this paper. Section 4 reports on a case study. Finally, section 5 closes this paper with some concluding remarks.

Short Papers of the 22nd IFIP ICTSS, Alexandre Petrenko, Adenilso Simao, Jose Carlos Maldonado (eds.), Nov. 08-10, 2010, Natal, Brazil.

2 The PRISM Language

A PRISM model is composed of one or more modules, each one having a set of finite state variables and its behavior being represented by one or more guarded commands in the following format:

```
[sync] <guard> -> p_1 : update_1 + ... + p_n : update_n;
```

When the **<guard>** expression is true, one or more transitions within the command are enabled, each one having an associated probability of occurrence **p_n** and an update expression **update_n** that assigns new values to one or more variables. Furthermore, modules can synchronize to each other if the same label [sync] is specified to two or more commands of distinct modules.

PRISM supports different types of probabilistic models. However, for this paper we are only concerned with Continuous-Time Markov Chain (CMTC) models. In a CMTC, the numeric parameter associated to each transition corresponds to a rate of negative exponential distribution. Unlike other model checkers, PRISM supports verification of quantitative properties. For this purpose the tool incorporates operators from PCTL [4] and CSL [5].

3 Proposed Approach

At the moment, our approach relies on hand coding of a system fault model using low-level features of PRISM, presuming that the initial concept of architecture exists. However, as future work we foresee that this task can be abbreviated by automatic generation of PRISM input models from an architecture model written in a high level description language more suitable for system designers.

Table 1. Fault Modelling Principles

Fault Type	Modeling
Permanent	Irreversible assignment to the fault variable
Transient	Reversible assignment to the fault variable
Dependant	Command synchronization
Degradation	Failure rate as a function of local or external fault variables

Starting from the system architecture concept, the fault behavior of each component is characterized. Potential faults are identified and the failure rates are estimated using manufacturer data or reliability estimation methods. With that information, each component is represented as a PRISM module following modeling principles summarized in Table 1.

Basically, each potential fault is modeled using a local variable combined with one or more commands that specifies: i) the expected rate and ii) new value for that variable to implement the required fault behavior. The following fragment of code exemplifies a component fault model that includes a permanent and a transient fault:

```
module Sys_Comp1
  c1_f1: [0..1] init 0; // Permanent fault variable
  c1_f2: [0..1] init 0; // Transient fault variable
  [] (c1_f1=0) -> (λc1_f1):(c1_f1'= c1_f1+1);
  [] (c1_f2=0) -> (λc1_f2):(c1_f2'= c1_f2+1);
  [] (c1_f2>0) -> (1):(c1_f2'= c1_f2-1);
endmodule
```

At the first lines local variables `c1_f1` and `c1_f2` are declared. The state of `c1_f1` is the guard for the first command line that performs an irreversible assignment to it, hence a permanent fault with an associated rate of λ_{c1_f1} per units of time. On the other hand, variable `c1_f2` represents a transient fault which occurs at a rate of λ_{c1_f2} but is recovered with a rate of 1.

The system fault model should also include the environment to account for external hazards that may cause or contribute to component faults. The code fragment bellow shows an environment model and a component that is affected by an external hazard:

```
module Environment
  eh1: [0..1] init 0;
  [eh1] (true) -> (λeh1) : (eh1'= eh1);
endmodule

module Sys_Comp2
  c2_f1: [0..1] init 0;
  [eh1] (c2_f1=0) -> (λc2_f1) : (c2_f1'= c2_f1+1)
                  + (1- λc2_f1) : (c2_f1'= c2_f1);
endmodule
```

The `Environment` module includes a single variable `eh1` to account for an external hazard, which can be for instance, a lightning strike, a radiation dose, and so on. Since command for `eh1` is not guarded, the transition is always enabled with a rate of λ_{eh1}. The transition associated to this command synchronizes with the command inside the `Sys_Comp2` by using same label `[eh1]`. Inside `Sys_Comp2`, two transitions that update `c2_f1` are possible: the first increments the variable by 1 in order to indicate that `eh1` has induced a failure, and the second, which does not change the value of `c2_f1`, represents the possibility that the system is not always affected by this hazard.

Finally, the code fragment shown bellow is the fault model that accounts for the reliability degradation of a system component:

```
module Sys_Comp3
  c3_f1: [0..1] init 0;
  [] (c3_f1=0) -> (λc2_f1 + c1_f1*λc2_f1) : (c3_f1'= c3_f1+1);
endmodule
```

This effect is easily represented in PRISM since rates can be dependant upon other variables. In the above example, the rate is a function of `c1_f1`, which indicates the occurrence of a fault in another component. This can represent, for instance, an unexpected rise of the system operation temperature.

Once created, the fault models of individual components can be reused in different evaluations and possibly, in different solutions. Furthermore, the fault model can be updated to reflect the best knowledge about the fault behavior of the architecture components.

4 Case Study

This section summarizes a simplistic case study whose purpose is to illustrate through a practical example the application of the principles described in Section 3. It is worthy mentioning that rates have been included as constants declared using reference values which are not shown. The actual values for these constants should be changed as more detailed reliability data is available and has no influence on the scope of this paper.

4.1 System Overview

The system of interest is a weapon management system (WMS) for military aircraft [6]. Fig. 1 shows the generic architecture for the WMS that contains a set of weapon carriers (X, Y and Z), a computer and a relay box, which provides electrical power and discrete control interfaces. The stores are controlled by the weapon carriers that communicate with the WMS computer through the WMS databus. Finally, the avionics databus integrates the WMS computer with other aircraft systems.

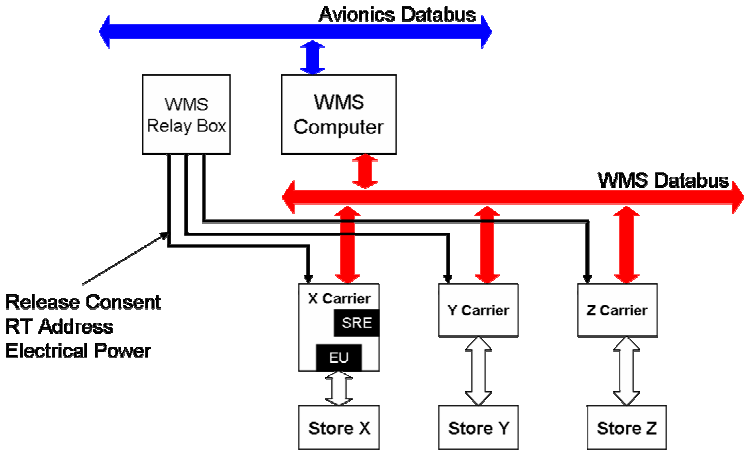


Fig. 1. Generic architecture of WMS for modern fighter aircraft as per MIL-STD-1760.

Each weapon carrier has two main components: the Suspension and Release Equipment (SRE), an electromechanical equipment responsible for the ejection of the store, and the Electronic Unit (EU), a stand-alone computer unit which performs main functions of store configuration, arming and launching sequence.

4.2 Component Fault Models

Because of space limitations of this paper, only component fault models for the Databus, Relay Box and SRE will be described.

The following PRISM module encapsulates the fault model for the Databus:

```
module Databus
  i1_f1: [0..1] init 0; // Permanent loss of databus
  i1_f2: [0..1] init 0; // Transient loss of databus
  [](i1_f1=0) -> (rate_of_i1_f1) : (i1_f1'=i1_f1+1);
  [](i1_f2=0) -> (rate_of_i1_f2) : (i1_f2'=i1_f2+1);
  [](i1_f2>0) -> (1) : (i1_f2'=i1_f2-1);
endmodule
```

This fault model has a permanent and a transient fault. The first represents the possible permanent loss of the databus while the second accounts for a momentary fault resulting in the loss of one frame.

The WMS Relay Box is modeled as follows:

```
module Relay_Box
  i3_f1: [0..1] init 0;  // Permanent power shutdown
  [eh2](i3_f1=0) -> (k*rate_of_i3_f1):(i3_f1'=i3_f1+1)
                  + (1-k*rate_of_i3_f1):(i3_f1'=i3_f1);
  [](i3_f1=0) -> (rate_of_i3_f1):(i3_f1'=i3_f1+1);
endmodule
```

In this model a single permanent fault represents an electrical power shutdown. This fault may happen with at a specific rate, or it can be induced by the external hazard `eh2` when the probability of occurrence is increased by a `k` factor.

Lastly, the following is the fault model for the SRE:

```
module Carrier_SRE
  c2_f1:  [0..1] init 0; // Permanent loss of firing capability
  c2_f2:  [0..1] init 0; // Transient self-activation
  [](c2_f1=0) -> (rate_of_c2_f1):(c2_f1'=c2_f1+1);
  [](c2_f1>0) -> (1):(c2_f1'=c2_f1-1);
  [eh1](c2_f2=0) -> (k*rate_of_c2_f2):(c2_f2'=c2_f2+1)
                  + (1- k*rate_of_c2_f2):(c2_f2'=c2_f2);
endmodule
```

The SRE fault model considers a permanent fault, which represents a safe failure of the internal explosive load, and a transient fault, which considers the possibility of self-activation, accounting for the possibility of unexpected initiation of the launching explosive. The SRE model is also sensitive to external hazard `eh1`. At the occurrence of `eh1`, the transient fault will occur with a rate increased by `k`.

4.3 Property Verification

Table 2. Property Specification in PCTL

	Property	**Result**
P1	$P<10^{-6}$ [true U[0,100] "Loss_of_firing"]	False
P2	$P<10^{-9}$ [true U[0,100] "Inadvertent_firing"]	True

Table 2 shows two examples of proprieties that were specified in the scope of this case study. In an industrial context, these properties may derive from the safety assessment process during which safety requirements are defined.

One may observe that labels have been used to facilitate understanding. These labels were included in the model and accounts for an expression that represents the desired condition. Label `Loss_of_firing` for instance, has been declared as follows:

```
label "Loss_of_firing" = i1_f1=1 | i2_f2=1 | i3_f1=1 | c2_f1=1;
```

Property P1 states that the probability of a "loss of firing" capability failure is less than 10^{-6}. On the other hand, property P2 states that the probability of an "inadvertent firing" failure is less than 10^{-9}. The last column provides the results the model checker returned.

5 Final Remarks

This paper has considered probabilistic model checking for the safety analysis of complex airborne systems. We have showed how to create a system fault model as a combination of fault models for its individual components, taking into consideration the external environment. In comparison to related work, the main advantage of this approach relies on the fact that quantitative aspects can be taken into account.

We understand that probabilistic model checking can provide effective means to identify potential flaws and to evaluate design trade-offs with respect to reliability and safety aspects at early phases of development. As future work, we foresee the automatic generation of PRISM models from a high-level architectural description language such as AADL. This would represent a step ahead towards the introduction of probabilistic model checking in the industrial context.

References

1. Marco Bozzano, Antonella Cavallo, Massimo Cifaldi, Laura Valacca and Adolfo Villafiorita. Improving Safety Assessment of Complex Systems: An industrial case study. Lecture Notes in Computer Science. Vol. 2805/2003. pages 208-222.
2. Anjali Joshi, Steven P. Miller, Michael Whalen and Mats P.E. Heimdahl. A Proposal for Model-Based Safety Analysis.
3. Marta Kwiatkowska, Gethin Norman and David Parker. PRISM: Probabilistic Model Checking for Performance and Reliability Analysis. ACM SIGMETRICS Performance Evaluation Review. March 2009.
4. H. Hansson and B. Jonsson. A logic for reasoning about time and reliability. Formal Aspects of Computing, 6(5):512-535, 1994.
5. A. Aziz, K. Sanwal, V. Singhal, and R. Brayton. Verifying continuous time Markov chains. In Proc. CAV'96, volume 1102 of LNCS, pages 269-276. Springer, 1996.
6. Matthew John Squair. Safety, Software Architecture and MIL-STD-1760. Proceedings of the 11th Australian Workshop on Safety Critical Systems and Software. Vol. 69. pages 93-112, 2007.

Learning Finite State Models of Observable Nondeterministic Systems in a Testing Context

Khaled El-Fakih[1], Roland Groz[2], Muhammad Naeem Irfan[2] and Muzammil Shahbaz[3]

[1] American University of Sharjah
P. O. Box 26666 Sharjah, UAE
kelfakih@aus.edu
[2] Grenoble Universities
38402 Saint Martin d'Hères, France
{Irfan, Groz}@imag.fr
[3] Fraunhofer IESE
67663 Kaiserslautern, Germany
Muzammil.Shahbaz@iese.fraunhofer.de

Abstract. Learning models from test observations can be adapted to the case when the system provides nondeterministic answers. In this paper we propose an algorithm for inferring observable nondeterministic finite state machines (ONFSMs). The algorithm is based on Angluin *L** algorithm for learning DFAs. We define rules for constructing and updating learning queries taking into account the properties of ONFSMs. Application examples, complexity analysis and an experimental evaluation of the proposed algorithm are provided.

Keywords: Nondeterministic finite state machine, inferring algorithm, conformance testing.

1 Introduction

Inferring finite state models is an old yet still active research activity with many relevant application domains. In the last decade, it has in particular been applied in the field of software system testing [5]. Angluin [1] proposed an algorithm, called L^*, that learns a minimal DFA accepting the target regular language with a polynomial number of queries (wrt the size of alphabet and state space of the automaton). This algorithm consists in asking two types of queries, in particular, *membership* and *equivalence* queries. A membership query is asked by the learner to test whether a certain string over the given alphabet is actually in the given language. An equivalence query is asked by the learner to test whether a conjectured model is correct (i.e. equivalent or has the same language as the given language). In testing software systems, membership queries can be more adequately replaced by output queries. An output query consists in testing a system with a given sequence of inputs and observing the corresponding sequence of outputs.

Nondeterminism occurs due to various reasons such as performance, flexibility, limited controllability and abstraction. In this paper, we present an algorithm with polynomial complexity for learning observable nondeterministic finite state machines

Short Papers of the 22nd IFIP ICTSS, Alexandre Petrenko, Adenilso Simao, Jose Carlos Maldonado (eds.), Nov. 08-10, 2010, Natal, Brazil.

(ONFSM). This class corresponds to machines whose outputs to a given input string are nondeterministic, but whose state is uniquely determined by the observation of (inputs and) outputs (this type of nondeterminism is sometimes called output-nondeterminism). Since an ONFSM is structurally equivalent to a DFA on the alphabet of input/output couples, we compare the new algorithm with DFA inference. We also discuss the implications of implementing membership and output queries on output-nondeterministic systems. We rely on the "all-weather conditions" assumption [3]. To implement membership queries on I/O systems, we use filters defined by Niese [4], which take into account the fact that an output query can answer many membership queries at once.

2 Preliminaries

A *finite state machine* (FSM), simply called a *machine* throughout the paper, is a 5-tuple $N = \langle Q, I, O, h_Q, q_0 \rangle$, where Q is a finite nonempty set of states with q_0 as the initial state; I and O are input and output alphabets; and $h_Q \subseteq Q \times I \times O \times Q$ is a behavior relation. A nondeterministic FSM (NFSM) is *observable* if for each pair $(q, i) \in Q \times I$ and each output o there is at most one state $q' \in Q$ such that $(q, i, o, q') \in h_Q$. In this paper, we consider only complete (for each pair *(q,i)* there is a corresponding transition in h_Q) observable nondeterministic FSMs, or *ONFSM* for short. Fig 1 provides a small example for illustration.

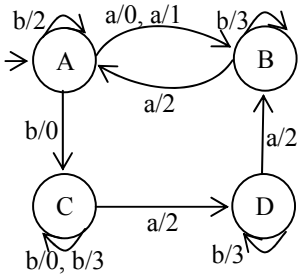


Fig 1. ONFSM over Q = {A,B,C,D}, I={a,b} and O={0,1,2,3}.

The different types of queries that will be used in the paper are defined as follows. For a DFA accepting an unknown language L over a given alphabet Σ, a *membership query* is a string $\alpha \in \Sigma^*$ to test whether α is in L. The output is either "yes" if $\alpha \in L$, or "no" otherwise. An *equivalence query* is a conjectured language L' and the output is "yes" if $L' = L$, or "no" otherwise. If the answer is no, then a counterexample α is returned, where α belongs to the symmetric difference of L and L'.

3 Inference of ONFSMs

Table 1. Initial observation table for ONFSM in Fig1.

		E	
		a	**b**
S_S	**ε/ε**	0, 1	0, 2
S_P	**a/0**	2	3
	a/1	2	3
	b/0	2	0,3
	b/2	0,1	0,2

An observation table consists of rows indexed by a prefix closed set Γ of I/O sequences, i.e., $\Gamma \subseteq (I \times O)^*$, columns indexed by suffix closed set E of input strings, and the content of a cell at the intersection of a row $\gamma \in \Gamma$ and a column $e \in E$ will be defined by $T(\gamma, e)$ belonging to the power set of O^+. A member of Γ is usually referred as α/β, or γ in short. We further restrict OT by defining $\Gamma = S_S \cup S_P$, such that $S_S \subseteq (I \times O)^*$ and $S_P = \{(\alpha i/\beta o): o \in T(\alpha/\beta, i)\}$, $\forall (\alpha/\beta \in Ss) \wedge (i \in I)$.

We define the equivalence of rows in OT with the help of the function T as follows. Two rows $\gamma_1, \gamma_2 \in \Gamma$ are *equivalent*, denoted by $\gamma_1 \cong \gamma_2$, iff $\forall e \in E$, $T(\gamma_1, e) = T(\gamma_2, e)$. A table is *closed* iff for each $\gamma_1 \in S_P$ there exists $\gamma_2 \in S_S$, such that $\gamma_1 \cong \gamma_2$. When the table is closed, then a complete *ONFSM* conjecture $C = \langle Q', I, O, h'_O, q'_0 \rangle$ can be constructed from the table as follows:

Definition 1 (Conjecture):

- $Q' = S_S$, i.e., the strings in S_S are the states of the conjecture
- $q'_0 = \varepsilon/\varepsilon \in S_S$ is the initial state of the conjecture
- For each $\alpha/\beta \in S_S$, $i \in I$ and $o \in T(\alpha/\beta, i)$; add $(\alpha/\beta, i, o, \gamma)$ into h'_O, where $\gamma \in S_S$ such that $\gamma \cong \alpha i/\beta o$

Theorem 1: If (Γ, E, T) is a closed observation table, then the conjecture $C = \langle Q', I, O, h'_O, q'_0 \rangle$, constructed as defined in Definition 1, is consistent with the finite function T. That is, for every $\alpha/\beta \in S_S$ and $e \in E$ such that $T(\alpha/\beta, e) = \{\beta_1, \ldots, \beta_d\}$, we have $(q'_0, \alpha e, \beta\beta_1, q'_1), \ldots, (q'_0, \alpha e, \beta\beta_d, q'_d) \in h'_O$.

Function Make-Closed(Γ, E, T)
Input: Observation table(Γ, E, T)
Output: Observation table(Γ, E, T)
Begin

Find $\gamma_1 \in S_P$ such that $\gamma_1 \ncong \gamma_2$, for all $\gamma_2 \in S_S$ and move γ_1 from S_P to S_S.
Extend (Γ, E, T) by adding $\alpha i/\beta o$, $\forall i \in I \wedge o \in T(\gamma_1 = \alpha/\beta, i)$, to S_P.
Ask output queries for the added rows in S_P and update (Γ, E, T) as follows:
$\forall$ added row $\alpha i/\beta o$ and $\forall e \in E$, ask output queries $\alpha i e$.
Update $T(\alpha i/\beta o, e)$ accordingly.

End

Algorithm 1: The algorithm L_N for inferring a complete ONFSM
Input: Black-box (unknown) complete ONFSM $N = \langle Q, I, O, h_O, q_0 \rangle$
Output: A complete minimal ONFSM Conjecture $C = \langle Q', I, O, h'_O, q'_0 \rangle$ that is equivalent to N
Begin
Initialize (Γ, E, T) with $S_S = \varepsilon/\varepsilon$, $S_P = \varnothing$, $\Gamma = S_S \cup S_P$, and $E = I$.
Ask output queries from (Γ, E, T) and update (Γ, E, T) as follows:
$\forall\ \alpha/\beta \in \Gamma, e \in E$, apply the output query $\alpha \cdot e$ to N.
Update $T(\alpha/\beta, e) = \{\beta_1, \ldots, \beta_d\}$, where $(q_0, \alpha, \beta, \gamma) \in h_O$ and $(\gamma, e, \beta_1, \gamma_1'), \ldots, (\gamma, e, \beta_d, \gamma_d') \in h_O$.
Update S_P as $S_P = \{i/o \mid i \in I \text{ and } o \in T(\varepsilon/\varepsilon, i)\}$ and complete (Γ, E, T) by asking output queries for all the rows in S_P.
while (Γ, E, T) is not closed
Make-Closed(Γ, E, T)
endwhile
Make the ONFSM conjecture C (as described in Definition 1)
Ask the equivalence query for C to the oracle
repeat
if the oracle replies with a counterexample α/β **then**
while α/β is a counterexample
for $j = 2$ **to** $|\alpha|$ **loop**
if $suffix^j(\alpha) \notin E$ **then**
add $suffix^j(\alpha)$ to E
construct the output queries for the new column
fill (Γ, E, T) by asking output queries
if (Γ, E, T) is not closed **then break** for loop **endif**
endif
endfor
while (Γ, E, T) is not closed
Make-Closed(Γ, E, T)
endwhile
make the conjecture C
endwhile
endif
until the oracle says "yes" to the conjecture C
Return the conjecture C from (Γ, E, T).
End

Theorem 2: If (Γ, E, T) is a closed observation table and the conjecture C is constructed, as defined in Definition 1, has n states, then any other complete ONFSM C' that has n or fewer states and also consistent with T is isomorphic to C.

4 Example

As an application example of the algorithm L_N, consider the "unknown" ONFSM N shown in Fig. 1 with four states labeled A (initial state), B, C, and D, and defined over inputs $\{a, b\}$ and outputs $\{0, 1, 2, 3\}$.

Table 2. Making the table 1 closed by moving the rows *a/0* and *b/0* to S_S

	a	**b**
ε/ε	0, 1	0, 2
a/0	2	3
b/0	2	0,3
a/1	2	3
b/2	0,1	0,2
aa/02	0,1	0,2
ab/03	2	3
ba/02	2	3
bb/00	2	0,3
bb/03	2	0,3

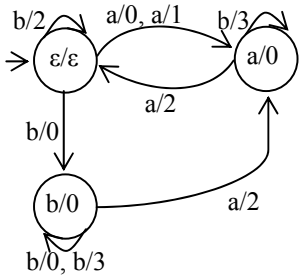


Fig 2. First ONFSM Conjecture

Table 3. Processing the counterexample *baab/0223*

	a	**b**	**ab**
ε/ε	0, 1	0, 2	03, 13
a/0	2	3	20, 22
b/0	2	0,3	23
a/1	2	3	22, 20
b/2	0,1	0,2	03, 13
aa/02	0,1	0,2	03, 13
ab/03	2	3	22,20
ba/02	2	3	23
bb/00	2	0,3	23
bb/03	2	0,3	23

Table 4. Making the table closed by adding the row *ba/02* to S_S

	a	b	ab
ε/ε	0, 1	0, 2	03, 13
a/0	2	3	20, 22
b/0	2	0,3	23
ba/02	2	3	23
a/1	2	3	22, 20
b/2	0,1	0,2	03, 13
aa/02	0,1	0,2	03, 13
ab/03	2	3	22,20
bb/00	2	0,3	23
bb/03	2	0,3	23
baa/022	2	3	22, 20
bab/023	2	3	23

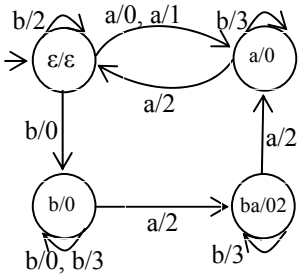


Fig 3 ONFSM Conjecture Table 4

5 Comparison of Algorithms

To have a significant set of experiments and assess the impact of various factors on the complexity (in particular $|I|$, n, m and d), we used large sets of randomly generated machines. Although L_N and L^* use different structures for observation tables, the "book-keeping" implied by the algorithms is overshadowed by the actual cost of interacting with black-box system, especially when testing nondeterministic systems.

When interacting with nondeterministic I/O systems, we may get different answers for the same input sequence. With the all-weather assumption, each output query is tried a number of times on the system, and the driver reports the set of all possible outputs. The cost of repeating the test may depend on the length of the query, but it

does not depend on the algorithm, so it will not be taken into account. The degree d of nondeterminism is defined as the average number of outgoing transitions for a given state and input. It is 1 for a deterministic machine. We will only consider values of d close to 1. For L^*, we use filters for membership queries [4], and *Suffix1by1* method to process the counterexamples [2] is used for both L^* and L_N algorithms. However, in Fig 2 data for L^* is also presented for Angluin method of processing counterexamples.

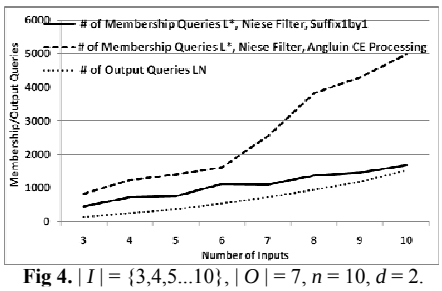


Fig 4. $|I| = \{3,4,5...10\}$, $|O| = 7$, $n = 10$, $d = 2$.

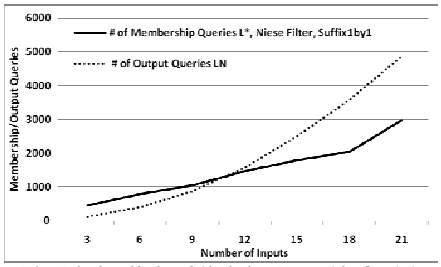


Fig 5. $|I| = \{3,6,...,21\}$, $|O| = 5$, $n = 10$, $d = 1.1$.

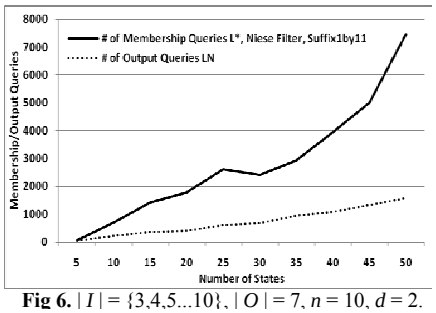


Fig 6. $|I| = \{3,4,5...10\}$, $|O| = 7$, $n = 10$, $d = 2$.

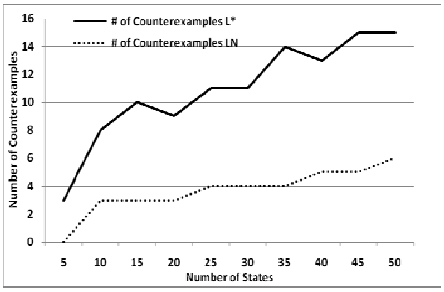


Fig 7. $|I| = 3$, $|O| = 2$, $n = \{5,10,15,...50\}$, $d = 1.1$.

6 Conclusion

Given a black-box (or unknown) ONFSM, the algorithm L_N infers an ONFSM that is equivalent to the unknown ONFSM via queries obtained using the "all-weather conditions" assumption. In most cases, L_N provides a significant improvement in performances when compared with approaches based on learning a DFA equivalent.

Possible extensions of this work involve learning NFSMs when "all-weather conditions" assumption cannot be satisfied, e.g. with limited controllability over SUT [6]. In this case learning has to be carried out using not the equivalence relation but other relations defined for NFSMs such as the separability and reduction relations.

References

1. D. Angluin: Learning Regular Sets from Queries and Counterexamples. Information and Computation: 2, 87--106 (1987).
2. M. N. Irfan, C. Oriat, R. Groz: Angluin Style Finite State Machine Inference with Non-optimal Counterexamples, Workshop on Model Inference In Testing 2010, ISSTA, Trento, Italy, 11-19 (2010).
3. R. Milner: A Calculus of Communicating Systems, Springer Verlag, ISBN 0-387-10235-3. (1980).
4. O. Neise. An Integrated Approach to Testing Complex Systems. Ph.D. Thesis, University of Dortmund (2003).
5. H. Raffelt, M: Merten, B. Steffen and T. Margaria: Dynamic Testing via Automata Learning. In International Journal on Software Tools for Technology Transfer: 307--324 (2009).
6. N. Shabaldina, K. El-Fakih, N. Yevtushenko: Testing Nondeterministic Finite State Machines with Respect to the Separability Relation. In TestCom/FATES: 305--318 (2007).

Assume-guarantee reasoning with ioco testing relation

Laura Brandán Briones

Fa.M.A.F. Universidad Nacional de Córdoba, Argentina

Abstract. Compositional reasoning is typically based on assume-guarantee reasoning principles, which consider each component separately and take into account assumptions about the context of the component. This paper presents a combination of the assume-guarantee paradigm and **ioco**, a formal conformance relation for model-based testing that works on input-output transition systems (IOTS). We show that, under certain restrictions, assume-guarantee reasoning can be applied in the **ioco** context, enabling to check **ioco**-conformance by testing components' system separately. We improve on previous results, where specifications are required to be given as components, allowing the specifications to be complete systems. Moreover, we prove that assume-guarantee reasoning can also be applied even when hiding internal communication between components.

1 Introduction

Conformance testing tries to identify if a system under test (known as SUT) behaves as expected. We consider a black-box conformance testing framework to test components of a system, where both the components and their specifications are modeled as input-output transition systems (IOTS), which formalize system descriptions that interact with their environment by receiving inputs and offering outputs. An IOTS that has the input set I and the output set U is denoted as IOTS(I, U). The **ioco** input output conformance relation asserts when an implementation behaves as expected by a given specification, both modeled by two IOTS(I, U); **ioco** checks the inclusion of specification' Straces (that is, traces with information about *quiescence*); in the presence of non-determinism, **ioco** can distinguish systems that are indistinguishable under trace inclusion.

A previous approach [6] studies compositional properties of **ioco**. Given two pairs of *input-enabled* (defined below) systems i_1, s_1 as IOTS(I_1, U_1) and i_2, s_2 as IOTS(I_2, U_2) the following compositional rule is proved: if i_1 **ioco** s_1 and i_2 **ioco** s_2 hold then $i_1||i_2$ **ioco** $s_1||s_2$ holds. In this previous framework the global specification is given as a composition of two systems $(s_1||s_2)$ and required to be input-enabled. However, we believe that it is often natural and easier to initially write a global system specification as a single system (rather than to decompose it into subsystems s_1 and s_2). Also, it is often the case that components conform to their specifications only in specific *contexts* (or environments). We thus solve the following problem: is it possible to use formal testing in a compositional way, using a global specification and taking into account *assumptions* about the environments in which the components are supposed to operate?

Short Papers of the 22nd IFIP ICTSS, Alexandre Petrenko, Adenilso Simao, Jose Carlos Maldonado (eds.), Nov. 08-10, 2010, Natal, Brazil.

We combine **ioco** with assume-guarantee reasoning, a "divide-and-conquer" approach that infers global system properties by checking individual components in isolation, under environment assumptions [2–4]. Assuming A, we prove that:

$$\frac{i_1||A \textbf{ ioco } S \quad \wedge \quad i_2 \textbf{ ioco } A}{i_1||i_2 \textbf{ ioco } S}$$

As a consequence, if A is provided we can test in isolation components (which may be at different stages of development and possibly written by different developer teams), having a specification of the overall system. Moreover, we prove that if the communication between the systems is hidden, a similar result also holds:(**hide** V **in** $i_1||A$) **ioco** S and i_2 **ioco** A then (**hide** V **in** $i_1||i_2$) **ioco**S.

2 The ioco testing relation & composition in IOTS

This section recalls same aspects of the **ioco** theory, for more details see [5].

An input-output transition system ($IOTS$) is a tuple $\langle Q, q^0, L, T\rangle$, where: Q is a countable, non-empty set of states, with $q^0 \in Q$ the initial state. L is a countable set of labels, partitioned into input (I) and output (U) actions, with $I \cap U = \emptyset$ and $I \cup U = L$. $T \subseteq (Q \times (L \cup \{\tau\}) \times Q)$ is the transition relation.

We denote the class of all labeled transition systems over I and U by $IOTS(I, U)$. We use a special label $\tau \notin L$ to denote internal actions. For an arbitrary $L' \subseteq L$, we use L'_τ as a shorthand for $L' \cup \{\tau\}$. For a system p, we write Q_p, L_p, and so on to denote the components of p. We write $q \xrightarrow{\mu} q'$ when $(q, \mu, q') \in T$. We use "?" and "!" before a label to denote whether it is an input or output action. A state that cannot perform an internal action is called *stable*, whereas one that cannot do an output or internal action is called *quiescent*. We use the symbol δ (with $\delta \notin L_\tau$) to represent quiescence. For an arbitrary $L' \subseteq L_\tau$, we use L'_δ as shorthands for $L' \cup \{\delta\}$. An $IOTS$ is called *strongly convergent* if it does not have infinite τ-labeled sequence. The set of all traces over L' (with $L' \subseteq L$) is denoted by L'^*; a trace in L'^* is denoted by σ, with ϵ denoting the empty sequence. If $\sigma_1, \sigma_2 \in L'^*$, then $\sigma_1 \cdot \sigma_2$ is the concatenation of σ_1 and σ_2. We use the standard notation with single and double arrows for traces: $q \xrightarrow{\mu_1 \cdots \mu_n} q'$ denotes $q \xrightarrow{\mu_1} \cdots \xrightarrow{\mu_n} q'$, $q \stackrel{\epsilon}{\Rightarrow} q'$ denotes $q \xrightarrow{\tau \cdots \tau} q'$ and $q \stackrel{\mu_1 \cdots \mu_n}{\Longrightarrow} q'$ denotes $q \stackrel{\epsilon}{\Rightarrow}\xrightarrow{\mu_1}\stackrel{\epsilon}{\Rightarrow} \cdots \stackrel{\epsilon}{\Rightarrow}\xrightarrow{\mu_n}\stackrel{\epsilon}{\Rightarrow} q'$, with $\mu_i \in L'_{\tau\delta}$. We write $q \stackrel{\mu}{\Rightarrow}$ if there exists a q' such that $q \stackrel{\mu}{\Rightarrow} q'$. An IOTS$(I, U)$ system $p = \langle Q, q^0, L, T\rangle$ is called *input-enabled* (denoted IOTS–**ie**) if all inputs are enabled in all states, i.e. $\forall\, q \in Q : \forall \mu \in I : q \stackrel{\mu}{\Rightarrow}$. Finally, for $I' \subseteq I_p$ and $\forall\, q \in Q : \forall \mu \in I' : q \stackrel{\mu}{\Rightarrow}$ we say that p is input-enabled with respect to I', denoted p-**ie**(I'). Moreover, as in typical process algebra semantics, we sometimes use a transition system and its initial state interchangeably.

Let p be an IOTS(I, U), $Q' \subseteq Q_p$ a subset of states in p, $q \in Q_p$ and $\sigma \in L^*_\delta$, then: q **after** $\sigma \triangleq \{q' | q \stackrel{\sigma}{\Rightarrow} q'\}$, **out**$(q) \triangleq \{\mu \in U | q \xrightarrow{\mu}\} \cup \{\delta | q \xrightarrow{\delta}\}$, **out**$(Q') \triangleq \bigcup\{\textbf{out}(q) | q \in Q'\}$ and Straces$(p) \triangleq \{\sigma \in L^*_\delta | q^0_p \stackrel{\sigma}{\Rightarrow}\}$.

Definition 1. *Given implementation $i \in IOTS(I, U)$-**ie** and specification $S \in IOTS(I, U)$: i **ioco** S iff* $\forall\, \sigma \in Straces(S) : \textbf{out}(i \textbf{ after } \sigma) \subseteq \textbf{out}(S \textbf{ after } \sigma)$.

Restricted ioco Since the **ioco**-relation gives implementation's freedom on inputs that do not appear in specifications, it is natural to ask if **ioco** holds when the specification inputs are included in the implementation inputs.

Definition 2. *Let* $p = \langle Q_p, q_p^0, L_p, T_p \rangle$ *be an IOTS*(I_p, U_p) *with* $L_p = I_p \cup U_p$, $I \subseteq I_p$ *and* $U \subseteq U_p$ *we define the restriction of* p *in* (I, U)*, denoted by* **R-**$p(I, U)$*, as the IOTS defined by* $Q = Q_p$, $q^0 = q_p^0$, $L = I \cup U$ *and* $T = T_p \cap (Q_p \times L_\tau \times Q_p)$.

In the case that the implementation actions are a subset of the specification actions, the **ioco** relation restricted to the specification actions follows easily (In the statement of the lemma below, we slightly abuse notation w.l.o.g. and assume that **ioco** allows inputs of the specification to be included, and not exactly equal, to the inputs of the implementation).

Lemma 1. *Let* i_1 *be an IOTS*(I_{i_1}, U_{i_1}) *and* s *be a IOTS*(I_s, U_s) *with* $I_s \subseteq I_{i_1}$ *and* $U_{i_1} \subseteq U_s$*. Let* i_2 *be the restriction of* i_1 *in* (I_s, U_s)*,* $i_2 =$ **R-**$i_1(I_s, U_s)$*, with* i_2-**ie**(I_s) *then: if* i_2 **ioco** s *then* i_1 **ioco** s

One of the advantages of the **ioco** relation is that it gives freedom to implementation behaviors that are not specified by the specification[5]. As a result, in Lemma 1, we use this liberty to test implementations with a wider input actions set than the ones specified by the specification. Also, we relax the *input-enabled* assumption to the subset of specification's input actions.

2.1 Composition in IOTS

The integration of components can be modeled algebraically by putting the components in parallel while synchronizing their common actions. The synchronization of processes p_1 and p_2 is denoted $p_1||p_2$.

Definition 3. *Let* $p_1 = \langle Q_1, q_1^0, L_1, T_1 \rangle$ *and* $p_2 = \langle Q_2, q_2^0, L_2, T_2 \rangle$ *be IOTS' with* $I_1 \cap I_2 = U_1 \cap U_2 = \emptyset$ *then* $p_1||p_2 = \langle Q, q_1^0||q_2^0, L, T \rangle$*, is defined as:* $Q = \{q_1||q_2 \mid q_1 \in Q_1, q_2 \in Q_2\}$, $I = (I_1 \setminus U_2) \cup (I_2 \setminus U_1)$, $U = U_1 \cup U_2$ *and* T *is the minimal set satisfying the following inference rules* ($\mu \in L_\tau$)*:*

$q_1 \xrightarrow{\mu} q_1', \mu \notin L_2 \vdash q_1||q_2 \xrightarrow{\mu} q_1'||q_2$; $q_1 \xrightarrow{?\mu} q_1', q_2 \xrightarrow{!\mu} q_2', \mu \notin \tau \vdash q_1||q_2 \xrightarrow{!\mu} q_1'||q_2'$

$q_1 \xrightarrow{!\mu} q_1', q_2 \xrightarrow{?\mu} q_2', \mu \notin \tau \vdash q_1||q_2 \xrightarrow{!\mu} q_1'||\ q_2'$; $q_2 \xrightarrow{\mu} q_2', \mu \notin L_1 \vdash q_1||q_2 \xrightarrow{\mu} q_1||q_2'$

Here, inputs $?a$ in one system are matched with outputs $!a$ in the other system, the result being an output $!a$ in their parallel composition. Given two systems p_1 and p_2, we use $Share(p_1, p_2)$ to denote $(I_1 \cap U_2) \cup (I_2 \cap U_1)$.

Note that Definition 3 puts constraints on input and output sets. So, parallel composition may give rise to IOTS' that are not *strongly convergent*, even if their components are. Thus, we implicitly restrict to cases where parallel composition is strongly convergent. Moreover, we only use binary parallel composition.

In Figure 1, on the left, we present three IOTS representing a cash machine. We represent the initial state with double circle. First, on the left hand side we present a device that takes care of the card received by our cash machine. The CARD device receives a card, announces that it has the card, and expects for an

OK answer or an error answer. If an OK answer arrives it gives back the card. If an error answer arrives the CARD device keeps the card, assuming something went wrong with the pin so the device retains the card.

Second, in the middle, we present a PIN device. It starts receiving the announcement that there is a card in the machine, then it expects a pin number. If the pin is correct, the device gives money and sends an OK acknowledgement. If the pin is incorrect, the device sends an error acknowledgement and an specific error stating that the pin was not correct.

Third, the CARD-PIN machine is depicted, that is the parallel composition of the CARD device and PIN device after the synchronization between them. In the CARD-PIN machine we see that the system receives a card and a pin number, then two things can happen. Either the system gives money and the card back (this is the case when the pin number was correct), or the system retains the card and gives an error indication that the pin was not correct.

3 Assume-guarantee reasoning with ioco

In this section we consider the following assume-guarantee rule: if $i_1||A$ **ioco** $S \ \wedge \ i_2$ **ioco** A holds then $i_1||i_2$ **ioco** S holds. The rule says that if, under assumption A, i_1 conforms with S and i_2 conforms A, then we can be sure that the parallel composition $i_1||i_2$ conforms w.r.t. S. We show (in Theorem 1 below) that under certain restrictions, this rule is sound. Therefore, we can obtain $i_1||i_2$ **ioco** S by checking $i_1||A$ **ioco** S and i_2 **ioco** A separately.

There are two complications in applying **ioco** in assume-guarantee reasoning. Firstly, *quiescent* states are not preserved by composition. Secondly, **ioco** allows implementation freedom for inputs that are not specified. To apply **ioco** we need to assume that implementations are input-enabled with respect to specification inputs. Therefore, to assert that $i_1||i_2$ **ioco** S, we also assume that $i_1||i_2$ is input-enabled with respect to S's inputs. We prove that, for an input-enabled system A such that $i_1||A$ **ioco** S and i_2 **ioco** A, then $i_1||i_2$ is **ioco** S. Note that we improve on previous result [6], and do not require S to be input-enabled.

Definition 4. *Let $\mu \in L_\delta$, $\sigma \in L_\delta^*$ and $L' \subseteq L_\delta$, then $\epsilon \lceil L' = \epsilon$ and $(\mu \cdot \sigma) \lceil L' = \sigma \lceil L'$ if $\mu \notin L'$ or $\mu \cdot (\sigma \lceil L')$ if $\mu \in L'$.*

In the proof of Theorem 1 we use the following result from [6]: Let p_1 be a $IOTS(I_1, U_1)$ and p_2 be a $IOTS(I_2, U_2)$ with $I_{p_1} \cap I_{p_2} = U_{p_1} \cap U_{p_2} = \emptyset$ and $L = I \cup U$ where $I = I_1 \cup I_2 / Share(p_1, p_2) \wedge U = U_1 \cup U_2$, $r \in Q_{p_1||p_2}, \sigma$ be in L_δ^*, then: $p_1||p_2 \stackrel{\sigma}{\Rightarrow} r$ iff $\exists\, p_1', p_2' : p_1 \stackrel{\sigma \lceil L_{p_1}^\delta}{\Longrightarrow} p_1' \wedge p_2 \stackrel{\sigma \lceil L_{p_2}^\delta}{\Longrightarrow} p_2' \wedge r = p_1'||p_2'$. Basically, if p is the parallel composition of p_1 and p_2 ($p = p_1||p_2$) under some restriction, for all reachable states r in p it is possible to find a state r_1 in p_1 and a state r_2 in p_2 such that $r = r_1||r_2$. The inverse also holds (see [6]).

Theorem 1. *Let i_1 be an $IOTS(I_1, U_1)$–**ie**(I_1), A and i_2 be $IOTS(I_2, U_2)$–**ie**(I_2) and S be an $IOTS(I_3, U_3)$. With $I_1 \cap I_2 = U_1 \cap U_2 = \emptyset$ and $U_1 \cup U_2 \subseteq U_3 \wedge I_3 \subseteq I_1 \cup I_2$ then: if $i_1||A$ **ioco** $S \ \wedge \ i_2$ **ioco** A hold then $i_1||i_2$ **ioco** S holds.*

Theorem 1 allows us to apply **ioco** not only to composed systems but also to specifications given as a complete system knowing an assumption about a subpart of the system. Compared to the previous result, the new assume-guarantee approach assumes that we can have an assumption on one of the components.

4 Hiding in assume-guarantee reasoning with ioco

Definition 5. *Let* $p = \langle Q, q^0, L, T\rangle$ *be an IOTS with* $L = I \cup U$ *and* $V \subseteq U$ *then we define* **hide** V **in** p *as a new IOTS with* $\langle Q', q'^0, L', T'\rangle$ *where:* $Q' = Q$, $q'^0 = q^0$, $labs' = I \cup (U \backslash V)$ *and* T' *is the minimal set satisfying the following inference rules, with* $\mu \in L_\tau$, $q_1, q_2 \in Q$ *and* $q_1', q_2' \in Q'$*:* $q_1 \xrightarrow{\mu} q_2$, $\mu \notin V \vdash q_1' \xrightarrow{\mu} q_2'$ *and* $q_1 \xrightarrow{\mu} q_2$, $\mu \in V \vdash q_1' \xrightarrow{\tau} q_2'$.

Again, note that Definition 5 gives constraints on the input and output sets. Therefore, hiding may gives rise to IOTS' that are not *strongly convergent*, even if their components are. So, we implicitly restrict ourselves to cases where hiding is *strongly convergent*.

On the right hand side in Figure 1 we show the H(CARD-PIN) machine, i.e. the parallel composition of the CARD and PIN machine after hiding the synchronization actions between them.

An immediate question, given from the results from Section 3, is the following: Is it possible to hide the internal communication between components? The answer is affirmative, as shown in the next theorem.

Theorem 2. *Let* $i_1 \in IOTS$–**ie**(I_1, U_1), $A, i_2 \in IOTS$–**ie**(I_2, U_2) *and furthermore* $S \in IOTS(I_3, U_3)$, *with* $I_1 \cap I_2 = U_1 \cap U_2 = \emptyset$. *Let* $V \subseteq Share(i_1, i_2)$ *and* $(U_1 \cup U_2)\backslash V \subseteq U_3 \wedge V \cap U_3 = \emptyset \wedge I_3 \subseteq I_1 \cup I_2$ *then: if* (**hide** V **in** $i_1||A$) **ioco** S *and* i_2 **ioco** A *hold then* (**hide** V **in** $i_1||i_2$) **ioco** S *holds.*

This result enables us to test the system allowing to incorporate new interfaces between components by only changing the assumption and leaving the specification of the whole systems the way it was.

Recall the specification S of the cash machine from Figure 1(CARD-PIN). There, we can insert a card, later a pin number, and obtain two answers from the machine: positive or negative. In the case we obtain a positive answer, we receive the money and the card. In the case we obtain a negative answer, we only receive a message informing that the pin number is wrong.

Now suppose two companies are developing the internal components of this cash machine; the two components interact as shown on the left of Figure 2. The cash machine has two internal components (CARD and PIN), which interact using the labels *have-card*, *OK* and *err*, but this is not specified in CARD-PIN. So, to allow both companies to test each part separately, the system that interacts with the CARD component is specified in A, shown in Figure 2.

Given one CARD implementation i_{CARD} and one PIN i_{PIN}, let V be the internal communication s.t. $V = \{\text{have-card}, \text{OK}, \text{err}\}$. Using Theorem 2, it is enough to prove that (**hide** V **in** $i_{\text{CARD}}||A$) **ioco** S and i_{PIN} **ioco** A, to establish that (**hide** V **in** $i_{\text{CARD}}||i_{\text{PIN}}$) **ioco** S.

Conclusions To the best of our knowledge, ours is the first application of assume-guarantee with **ioco**. We plan to extend this work considering probabilistic and realtime settings (e.g., the **tioco** timed testing relation [1]).

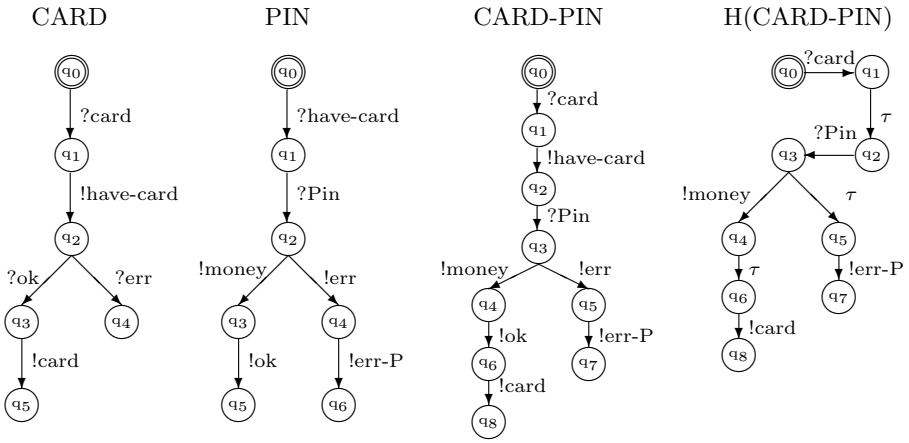


Fig. 1. Parallel composition of CARD and PIN and their hiding communication

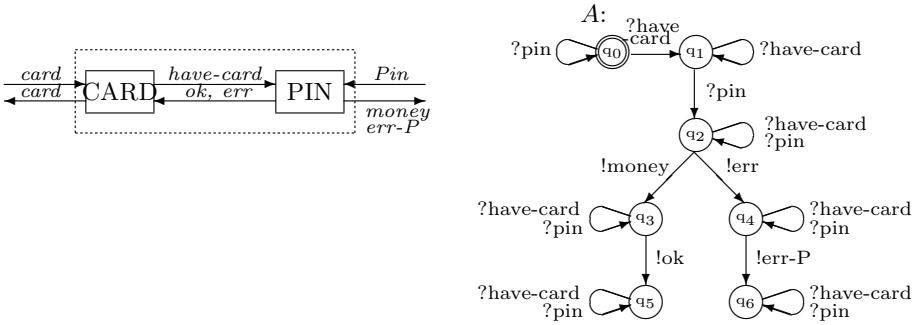


Fig. 2. Left: Internal components of the cash machine from Figure 1(left). Right: The assumption of the environment in which the CARD component is assumed to function

References

1. L. Brandán Briones and E. Brinksma. A test generation framework for quiescent real-time systems. In Brian Nielsen Jens Grabowski, editor, *Formal Approaches to Software Testing, FATES*, Linz, Austria, Sep 2004. Springer-Verlag GmbH.
2. E. M. Clarke, D. E. Long, and K. L. McMillan. Compositional model checking. In *Symposium on Logic in Computer Science*, June 1989.
3. W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes. Generating finite state machines from abstract state machines. In *ISSTA*, July 2002.
4. A. Pnueli. In transition from global to modular temporal reasoning about programs. In *K. Apt, editor, Logic and Models of Concurrent Systems, volume 13*, 1984.
5. J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. In *Software-Concepts and Tools, 17(3)*, pages 103–120, 1996.
6. Machiel van der Bijl, Arend Rensink, and Jan Tretmans. Component based testing with ioco. Technical report, University of Twente, 2003.

Test Driven Development with Oracles and Formal Specifications

Shadi Alawneh and Dennis Peters

Faculty of Engineering and Applied Science
Memorial University, St.John's, NL
Canada A1B 3X5
{shadi.alawneh,dpeters}@mun.ca

Abstract. The current industry trend to using Test Driven Development (TDD) is a recognition of the high value of creating executable tests as part of the development process. In TDD, the test code is a formal documentation of the required behaviour of the component or system being developed, but this documentation is necessarily incomplete and often over-specific. An alternative view to TDD is to develop the specification of the required behaviour in a formal notation as a part of the TDD process and to generate test oracles from that specification. In this paper we present tools in support of this approach that allow formal specifications to be written in a readable manner that is tightly integrated with the code through an integrated development environment, and test oracles to be generated automatically. The generated test code integrates smoothly with test frameworks (e.g., JUnit) and so can be directly used in TDD. This approach has the advantage that the specifications can be complete and appropriately abstract but still support TDD.

Keywords: Test Driven Development, Extreme Programming, Open Mathematical Documents, Test Oracle

1 Introduction

TDD is one of the core practices of Extreme Programming (XP). Two key principles of TDD are 1) that no implementation code is written without first having a test case that fails with the current implementation, and 2) that we stop writing the implementation as soon as all of the existing test cases pass.

In TDD, the test code is a formal documentation of some of the required behaviour of the system being developed. However, tests alone describe the properties of a program only in terms of examples and thus are not sufficient to completely describe the behaviour of a program. So, this documentation is necessarily incomplete and often over-specific. To solve this problem we propose an alternative approach to TDD, which is to develop a formal specification of the required behaviour as part of the TDD process and then generate test oracles from that specification. We thus propose a variation on the key TDD principles

Short Papers of the 22nd IFIP ICTSS, Alexandre Petrenko, Adenilso Simao, Jose Carlos Maldonado (eds.), Nov. 08-10, 2010, Natal, Brazil.

listed above: 1) No implementation code is written without first having a specification for the behaviour that is not satisfied by the current implementation, and 2) we stop writing the implementation as soon as the implementation satisfies the current specification. By generating oracles directly from the specification we are able to quickly and accurately check if the specification is satisfied by the implementation for the selected test cases.

We are using OMDoc (Open Mathematical Documents) [3] as a standardized storage and communications format for our specifications, and so we can take advantage of other tools.

2 Formal Software Specifications

Formal specifications are documentation methods that use a mathematical description of the software or hardware behaviour, which may be used to develop an implementation. With reference to the set of documents described in [6], in this work, we are focused on using module internal design documents [7] or module interface specifications to drive the development [8]. These two types of documents specify the behaviour of the module either in terms of the internal data structure and the effect of each access program on it, or in terms of the externally observable behaviour of the module.

There are several different formal methods for program specifications, e.g. the Java Modeling Language (JML) [4] and Z [9]. In our work, we use relations for the specifications, which characterize the acceptable set of outcomes for a given input. The specifications in our work consists of program specifications, which, in OMDoc terms, are symbol definitions contained within theories. Also, each symbol has a type and possibly other information. A program specification, describes the required behaviour of a program either in terms of the internal data structure and the effect of each access program on it, or in terms of the externally observable behaviour of the module. It consists of these components: constants; variables; auxiliary function and predicate definitions; the program invocation, which gives the name and type of the program and lists all its actual argument program variables; and an expression that gives the semantics of the program.

3 Oracle Generation

As described in [2], an oracle is some method for checking whether the system under test has behaved correctly on a particular execution.

In our work, an oracle is a program which, given a test input and output, will determine if it passes or fails with respect to the specification from which the oracle was derived. The oracle evaluates the characteristic predicate of the specification relation—if it evaluates to **true**, then that test input and output passes, otherwise it fails. Note that such an oracle does not require the existence of a correct version of the program.

Our tool can generate test oracles from both scalar expressions (logical operators, primitive relations, quantifications), and tabular expressions. Moreover, it

can handle auxiliary functions and predicates. It has been shown in the previous work that the tabular representation of relations and functions is a significant factor in making the documentation more readable, and so we have customized our tool to support them.

The oracle in our approach consists of two kinds of code: that generated by the Test Oracle Generator (TOG), and classes (e.g., Integer_Interval.Java, InvertedTable.Java, NormalTable.Java and VectorTable.Java), previously manually implemented that are used by the TOG generated code.

In the example described in Section 4.1, we are using one kind of tabular expression (Vector). Tabular expressions are implemented by instantiating an object of one of several classes of (Java) table objects which implement the various types of tabular expressions (normal, inverted and vector). These table objects contain all knowledge of the semantics of tabular expressions, so there is no need for this knowledge to be in the TOG. The expression in each cell of the table is implemented as Java class that extends CellBase and contains a procedure that evaluates that expression. This approach for implementing tabular expressions has the advantage that the oracle code can be more organized.

Table objects have a method, evaluateTable, which evaluates the tabular expression.

4 Test Driven Development with Oracles

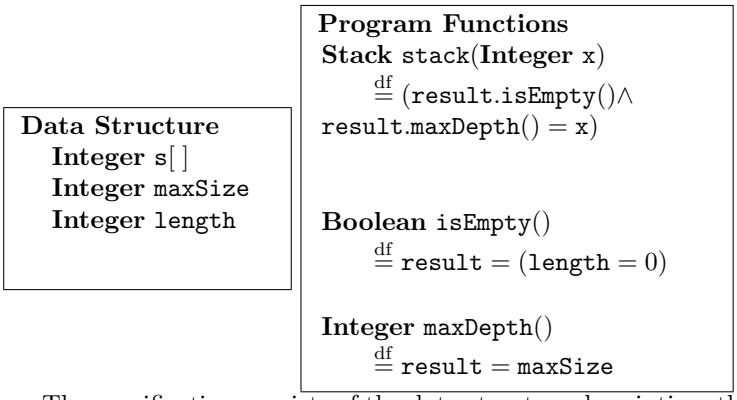
The work reported in this paper is an extension of the work reported in [1]. In [1], the test oracle generator supports generating test oracles from methods but in this work we extended the test oracle generator to allow the users to generate test oracles from module (class) specifications, which are based on the externally observable behaviour of the class. This will allow the use of oracles in class testing. Also, we have introduced an alternative approach to TDD that is to develop the specification of the required behaviour in a formal notation as a part of the TDD process and to generate test oracles from that specification.

The process looks like this:

- Write the specification for some required behaviour.
- Generate the test oracle from the specification and select test inputs.
- Run the program under test in the test framework (e.g., JUnit) using the test oracle to verify if it passes or fails.
- If the test fails, write code until this test passes.
- If the test passes and the specification is not completed yet, add to or refine the specification and redo the process again.
- We keep doing this process until the specification is complete.

4.1 Example

As an illustrative example we use the bounded stack as developed in [5]. The first step in our approach is to specify some required behaviour, in this case for creation of an empty stack:



The specification consists of the data structure description, the definition for `stack(x)` function, which is the program function specifying the behaviour of the constructor and two program functions specifying the behaviour of the methods `isEmpty()` and `maxDepth()`.

After we write the specification, we generate the test oracle from it and write the test code to call it (e.g., from JUnit). The test case will, of course, fail, so we should implement the constructor and methods so that the test case passes and we have a program that is consistent with the specified behaviour.

We then modify the specification for `push` to define behaviour for pushing on a non-full stack, and add three more methods:

void `push`(**Integer** `x`)
$\stackrel{df}{=}$

	$\texttt{p\_this.size}() \geq 0 \wedge \texttt{p\_this.size}() < \texttt{this.maxDepth}()$
$\texttt{this.size}() =$	$\texttt{p\_this.size}() + 1$
$\texttt{this}\vert$	$\forall i : [0, \texttt{p\_this.size}() - 1].($ $\texttt{this.elementAt}(i) = \texttt{p\_this.elementAt}(i)) \wedge$ $(\texttt{this.lastElement}() = \texttt{x})$

Integer `elementAt`(**Integer** `i`)
$\stackrel{df}{=}$ `result` = `s`[`i`]

Integer `size`()
$\stackrel{df}{=}$ `result` = `length`

Integer `lastElement`()
$\stackrel{df}{=}$ `result` = `s`[`length` − 1]

Here we use the naming convention of prepending "`p_`" to a program variable name (e.g., `p_this`) to represent the value of the program variable (e.g., `this`) in the state immediately before the function was executed. The new behaviour defined by the specification is to push an object on an a non-full stack. After the push the stack should contain that element and the size for the stack after is increased by one. Again we generate the test oracle and implement a test

case, which will initially fail. The stack code is then developed until the test case passes, and so it implements the specified behaviour.

Again we generate the test oracle and implement test cases, this time to push a few elements onto the stack.

Continuing in this manner, we eventually reach the full specification of the bounded stack, as below, and we have at the same time developed a full implementation and a full suite of test cases.

Data Structure

Integer s[ ]
Integer maxSize
Integer length

Program Functions

Stack stack(**Integer** x)
$\stackrel{\text{df}}{=}$ (result.isEmpty() $\wedge$ result.maxDepth() = x)

void push(**Integer** x)
$\stackrel{\text{df}}{=}$

	p_this.size() $\geq 0 \wedge$ p_this.size() $<$ this.maxDepth()	p_this.size() = this.maxDepth()
this.size() =	p_this.size() + 1	p_this.size()
this|	$\forall i : [0, \text{p\_this.size}() - 1].$ $\binom{\text{this.elementAt}(i) =}{\text{p\_this.elementAt}(i)} \wedge$ (this.lastElement() = x)	this = p_this

Integer pop()
$\stackrel{\text{df}}{=}$

	p_this.size() ≥ 1
this.size() =	p_this.size() $- 1$
this|	$\forall i : [0, \text{this.size}() - 1]. \binom{\text{this.elementAt}(i) =}{\text{p\_this.elementAt}(i)} \wedge$ (result = p_this.lastElement())

Integer top()
$\stackrel{\text{df}}{=}$ result = this.lastElement()

Boolean isEmpty()
$\stackrel{\text{df}}{=}$ result = (length = 0)

Integer maxDepth()
$\stackrel{\text{df}}{=}$ result = maxSize

Integer size()
$\stackrel{\text{df}}{=}$ result = length

Integer lastElement()
$\stackrel{\text{df}}{=}$ result = s[length $- 1$]

Integer elementAt(**Integer** i)
$\stackrel{\text{df}}{=}$ result = s[i]

5 Conclusions

In test driven development, tests are used to specify the behaviour of the program, and the tests are additionally used as a documentation of the program. However, tests are not sufficient to completely define the behaviour of a program because they only define the program behaviour by example and do not allow us to state general properties. So, the later can be achieved by using our TDD approach, which uses a formal specification to specify the behaviour of the program and supports testing directly against that specification by generating oracles.

Clearly a next step in this research and tool development will be to support test case generation from the specification as well, which will further reduce the amount of 'manual' test code development effort.

6 ACKNOWLEDGMENTS

This research was supported by the Faculty of Engineering and Applied Science at Memorial University and the Government of Canada through the Natural Sciences and Engineering Research Council (NSERC).

References

1. S. Alawneh and D. Peters. Specification-based test oracles with junit. Proc. IEEE Canadian Conference on Electrical and Computer Engineering (CCECE'10), Calgary, Canada, May. 2010.
2. W. E. Howden. *Functional Program Testing and Analysis.* McGraw-Hill Book Company, 1987.
3. M. Kohlhase. *OMDoc: An Open Markup Format for Mathematical Documents (Version 1.2).* Number 4180 in Lecture Notes in Artificial Intelligence. Springer Verlag, 2006.
4. G. T. Leavens, A. L. Baker, and C. Ruby. JML: a notation for detailed design. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications for Businesses and Systems*, chapter 12, pages 175–188. Kluwer, 1999.
5. J. Newkirk and A. Vorontsov. *Test-Driven Development in Microsoft .NET.* Microsoft Press, 2004.
6. D. L. Parnas and J. Madey. Functional documentation for computer systems. *Science of Computer Programming*, 25(1):41–61, Oct. 1995.
7. D. L. Parnas, J. Madey, and M. Iglewski. Precise documentation of well-structured programs. *IEEE Trans. Software Engineering*, 20(12):948–976, Dec. 1994.
8. C. Quinn, S. Vilkomir, D. Parnas, and S. Kostic. Specification of software component requirements using the trace function method. In *Int'l Conf. on Software Engineering Advances*, page 50, Los Alamitos, CA, USA, 2006. IEEE Computer Society.
9. J. M. Spivey. *The Z Notation: A Reference Manual.* International series in computer science. Prentice Hall, New York, 2nd edition, 1992.

Iterative Software Testing Process for Scrum and Waterfall Projects with Open Source Testing Tools Experience

Eliane Collins[1,2] ,Vicente Lucena [2]

[1] Instituto Nokia de Tecnologia - INdT,
Av. Torquato Tapajós, 7200 - Colônia Terra Nova- Manaus - Amazonas, Brasil
[2]Universidade Federal do Amazonas,
Av. Gal. Rodrigo Octávio Jordão Ramos, nº 3000– Manaus–Amazonas, Brazil
elianecollins@gmail.com, vicente@ufam.edu.br

Abstract. This paper describes a practical experience to implement and analyze an iterative software testing process using automated open source testing tools, within a project that used agile software development process Scrum and a project that used the classical Waterfall software development process. The software testing process applied in the modified development process provided improvement, stability and maturity to the developed software. The results obtained showed the benefits of an iterative testing process with automated testing activities for different projects processes ranging from a gain in test execution time and defects detection, to customer reliability for the final product without an increase in costs.

Keywords: software testing, test automation, open source testing tools.

1 Introduction

According to Myers [1], software testing is a process, or a series of processes, designed to make sure that computer code does what it was designed to do and that it does not do anything unintended. Software testing has become essential to the companies to ensure the product quality regardless of whether the development methodology. Software testing automation is one of the main approaches that have been applied to decrease testing costs and time while test automation requires automated test execution and results verification [2].

The classical Waterfall software development process is sequential, so there are well defined phases to execute the activities of requirements analyses, design, coding, integration tests and maintenance [3]. According to this development process, the system test activities must be done throughout the whole system in the test phase, but we know the defects may occur before this phase and treat them later will increase the project costs.

Therefore, contrasting to the classic process, the agile software development process is distinguished mainly because of its ability to rapidly accommodate changes in the original requirements and prioritize the functionalities development through

executable code instead of extensive written documentation output, and also quick responses to change and collaboration with the customer rather than following rigid plans and contract negotiations [4]. The agile processes are iterative, so the test activities follow the iterations and they may be executed efficiently and fast leading to the use of automated testing tools.

In this paper we will present experiences and results of the implementation of an iterative software testing process with open source testing tools for a project that used the agile process Scrum and another project that used the Waterfall process. For confidentially, the projects described in this paper will be called Project 1 and Project 2 respectively. In section 2, an explanation of the iterative test process proposed for the Scrum methodology is shown. In section 3, details on how the iterative test process was adapted to be executed in a Waterfall project will be commented. The section 4 presents the results and in section 5 the conclusion and lessons learned are going to be discussed.

2 Iterative Test Process for the Scrum Project

2.1 Scrum Project Environment Context

Scrum is an agile development process focusing on teamwork that can be used to manage and control software and products development using iterative and incremental practices [5].

The methodology controls the functionality to be implemented through a list called Product Backlog. For each sprint (iteration), there is a Sprint Planning Meeting where items on this list are prioritized by the customer (Product Owner). The team then defines the features that may be accomplished within the iteration. This list planned for the next iteration is called Sprint Backlog [5]. During the Sprint, the activities are broken into tasks that are observed by the team through a daily meeting which lasts a maximum of fifteen minutes where tasks impediments are identified.

The Project 1 is a system to provide a customer satisfaction survey result to the development team and also register the projects, leaders, customers, teams and the customers' answers. The project was developed under the Scrum process and used a web development platform: Ruby language, Rails framework [6], Aptana Studio as IDE (Integrated Environment Development) and MySQL database [11].

In the original Project 1 schedule there wasn't any time planned for test activities, which is indeed a recurrent mistake noticed in many software projects. In fact, the project had only one test analyst, a short time for test execution and the project had no financial resources available for this activity.

2.2 Software Testing Process Execution

According to the Project 1 environment, it wasn't possible to run a traditional testing process in time, so it was chosen to perform the software testing process main activities while minimizing on documentation to save time. The documents used were only the Test Plan, the Test Cases Specification and the Test Case Execution Report.

The tests were executed with free automated testing tools mainly for regression testing and with no additional cost.

The metric adopted for the tests was to cover 100% of the functional requirements described in the backlog. The tester also should use the exploratory testing technique, where the alternative system scenarios were explored to find navigational and performance defects.

The tester was a scrum team member and the sprint activities were broken into tasks to be executed in two weeks which were accompanied by the team in the daily scrum meeting. This allowed a greater interaction with the development team and any impediment to the progress of the task was identified very quickly having the whole development team involved.

At the first stage of the project, the tester chose free tools to automate the tests according to the project environment. Project 1 was a web application, so the tool of choice for the functional tests was the Selenium Core and Selenium IDE [7]. It is an open source tool that uses the approach rec-and-play support and test web applications for the browser Mozilla Firefox. The Selenium IDE can record the user actions in the browser, creating test scripts in HTML code and executing them later in the Selenium Core.

However, there was also the automation for managing, planning and preparing test cases in order to reduce the documentation time, since the results also had to be reported to the developers quickly. After in-depth research on available test tools, another open source tool called TestLink [8] was chosen. This tool can be used to write test cases and generate reports based on the test execution results. The tool adopted for defects registration was the Mantis [9], which is also open source and offers good resources for this kind of control and was already used in the company.

The test cycle was performed for each Sprint based on the following activities:

1. Plan the test cases for the Sprint stories according to the acceptance criteria and emphasizing test cases creation for exceptions. The system analyst should review the test cases to ensure the functionalities coverage;
2. Automated test scripts creation. The test scripts were implemented in the browser Mozilla Firefox, where the Selenium IDE works recording all the tester actions and turning them into HTML scripts. These scripts were edited and gathered in a test suite, and reused in other test cases;
3. In the Selenium Core tool, the test scripts were executed in Firefox and Internet Explorer browsers. The test cases execution identified application defects and the tester registered them in the TestLink tool. At Mantis, the defects were also registered, described and assigned to the developer responsible who was notified about the defect by e-mail;
4. The Reports were automatically generated. The TestLink offers several options to analyze the results based on test cases execution. The Reports used were: General Test Plan Metrics, Failed Test Cases, and Charts Total Bugs for Each Test Case. They can be accessed from the option Results from the top of the TestLink menu;
5. When the developer signaled in Mantis that defects had been fixed, the entire test suite was executed by Selenium validating defects and doing the regression test, ensuring that another part of the system was not affected by code changes;

6. If the acceptance criteria were met, the same activities process began again for the next Sprint.

The regression test execution for all browsers was fast because of the automated test scripts. At the end of the sprint, a summary defect report was generated with all the defects history in the system and presented to the product owner.

3 Iterative Test Process for a Waterfall Project

3.1 Waterfall Project Environment Context

The Project 2 was a software project to provide a graphical analysis of factory production test process in real time in order to advise professionals to support manufacturing on the quality trends for production line failures. The project development used the Waterfall process approach and the software platform used was: Java Swing, IDE Eclipse environment and Oracle database.

The Project 2 had twelve screens, where six of them were used to show and analyze graphical charts and the other six had the data inputs, reports and user configuration.

The project scenario when the test activities were requested was: outdated system requirements documentation, functionalities implemented but not tested, requirements changes not documented, delayed project schedule and just 4 weeks for a tester and a test analyst to do all test activities for the whole system.

3.2 Software Test Process Execution

Even with the critical project scenario, the test team has decided to adopt the same test process executed for the scrum project, but with some adaptations. The test goal was to ensure in a short time that at least the main features would be tested and stressed.

The first step was to attend a meeting with the development team to understand the project and verify the requirements documentation updates. After that, the test team asked for the prioritized list of the project features where the most important features had higher priority. According to this list, the test iterations were planned having the high priority features being tested first.

The test documents and metrics used were the same of the Project 1. The priority features should be 100% covered by tests for each iteration. The test team chose the automated testing tools according to the project platform. To automate the functional tests for java swing platform the tool chosen was Marathon Test Tool, which is an open source tool that uses the approach rec-and-play recording the user actions creating test scripts in Jython language and running them later in the tool [10]. For the test cases management and defect registration, the team still used the TestLink and Mantis as described in the section 2.2.

The test cycle was planned to be executed in four iterations and each iteration lasting one week. The testers must select only the feature with the highest priority to be tested in one week according to the list of the prioritized features. At the end of

each iteration, the development team released a new system version to be validated and the other features were planned and tested respecting the established priority order.

During the test iteration, the testers should do the following activities:

1. Plan the test cases for the priority features in the TestLink Tool. The system analyst should review the test cases to ensure the functionalities coverage;
2. Automated test scripts creation. The test scripts were implemented in Marathon test tool. The tester actions were recorded and turned into Jython scripts. These scripts were edited and gathered in a test suite, and reused in others test cases;
3. Then, the testers executed the automated tests in Marathon test tool and the exploratory manual tests too. The defects found were registered in Testlink and Mantis as described in section 2.2.
4. The automatic regression tests for all functionalities were executed also when defects were validated to ensure that another part of the system was not affected by code changes.
5. The automated TestLink reports were generated and sent to the project leader. The development team released another system version and the same activities process began again for the next test iteration.

At the end of the iterations, a summary defect report was generated with all the defects history in the system. This process allowed for the main features to be stressed.

4 Results

The test automation experience for Scrum Project 1 was very important. First, because the test activities became faster and 100% of the functionalities from backlog were covered with the tests. Then, the tester was able to find interface incompatibilities running the Selenium automated tests on different browsers, which would have taken too long if they had been done manually. It represented 65% of the system defects.

The regression tests that found defects in the functionalities that were considered done, were executed during each sprint. The customer did not find interface problems or functional defects, which earned a good evaluation of the project.

The test team, using the iterative software testing process for the Waterfall Project 2, initially, had the goal to ensure the test execution of at least the main features, but the automated tests execution allowed that all system requirements with tests be covered in four weeks. That was a very positive result as in a traditional test process, the execution of all test activities and its documentation would take more time and resources.

In four iterations, 77 defects were registered in Mantis, where 30% were interface defects and 70% were functional defects. The majority of the defects were found in regression tests because the code changes affected other features. The Project 2 had good acceptance from the client who decided to continue with the same team in the new project version.

5 Conclusion

Throughout the experiment, we observed some peculiarities in the different process methodologies that we take as lessons learned. In Project 1 the automated test scripts had to be reworked and updated with every backlog changes for each sprint. The automated test scripts code were reused, so the creation of the new test scripts has became faster. The scrum developer team has learned to estimate more time to defects fixing due to the large amount of problems found during the regression tests.

In the Project 2, the developer team took as lesson learned that they had to engage the test team from the inception of the project, because a lot of functional defects could have been found before the project implementation of the project. The test scripts creation has become faster because of the reuse of codes, which made it possible to do exploratory tests, where the usability problems were found. The test activities broken into iterations made it possible to stress the most important features, making the system attend to what was requested by the client.

From the projects presented we can conclude that the experience has shown good results allowing us to acquire knowledge in testing tools and testing processes, leading us to search for new automation tools for other development platforms for further projects in the company.

6 References

1. Myers, G.J.: The Art of Software Testing. Ed. John Wiley & Sons, Inc., Hoboken, New Jersey. (2004)
2. Shahamiri, S.R.: Kadir, W.M.N.W.; Mohd-Hashim, S.Z.: A Comparative Study on Automated Software Test Oracle Methods. In: ICSEA '09. Fourth International Conference, pp. 140 – 145, (2009).
3. Pressman, R.S.: Software Engineering. Ed. McGraw Hill. Pp. 38 – 39, (2006).
4. Taromirad, M.: Ramsin, R.: CEFAM: Comprehensive Evaluation Framework for Agile Methodologies, 2009. In: Software Engineering Workshop, 2008. SEW '08. 32nd Annual IEEE, pp. 195 – 204, (2008).
5. Hu Zhi-gen; Yuan Quan; Zhang Xi: Research on Agile Project Management with Scrum Method, 2009. In: Services Science, Management and Engineering, 2009. SSME '09. IITA International Conference on, pp. 26 – 29, (2009).
6. Viswanathan, V.: Rapid Web Application Development: A Ruby on Rails Tutorial, 2008. In: Software, IEEE, pp 98 – 106, (2008).
7. Holmes, A.; Kellogg, M.: Automating Functional Tests Using Selenium, 2006. In: Agile Conference, 6 pp. – 275. IEEE Press, (2006).
8. TestLink Testing Tool Documentation, http://www.teamst.org/ (2010).
9. Mantis Testing Tool Manual, http://www.mantisbt.org/ (2010).
10. Marathon Integrated Testing Environment, http://www.marathontesting.com (2010).
11. MySQL, http://www.mysql.com (2010).

List of Authors