MBT 2007
Third Workshop on
Model-Based Testing

March 31 - April 1, 2007
Braga, Portugal
Satellite workshop of ETAPS 2007

Organizers
Bernd Finkbeiner, Yuri Gurevich,
and Alexander K. Petrenko
Preface

This volume contains the proceedings of the Third Workshop on Model Based Testing (MBT 2007). MBT 2007 is to be held March 31 - April 1, 2007, as a satellite workshop of the European Joint Conferences on Theory and Practice of Software (ETAPS 2007).

The workshop is devoted to model-based testing of both software and hardware. Model-based testing uses models that describe behavior of the system under consideration to guide such efforts as test selection and test results evaluation. Model-based testing has gained attention with the popularization of models in software/hardware design and development. Not all the models used now are suitable for testing. Models with formal syntax and precise semantics are particularly important. Testing based on such a model helps one to measure to what degree the code faithfully implements the model.

Techniques to support model-based testing are drawn from diverse areas, like formal verification, model checking, control and data flow analysis, grammar analysis, and Markov decision processes. The intent of this workshop is to bring together researchers and users using different kinds of models for testing and to discuss the state of art in theory, applications, tools, and industrialization of model-based testing.

We would like to thank the program committee and all reviewers for the excellent job of evaluating the submissions. We are also grateful to the ETAPS 2007 organizing committee for their valuable assistance.

Bernd Finkbeiner, Yuri Gurevich, and Alexander K. Petrenko
February 2007
Program Committee

Bernhard K. Aichernig (Graz University of Technology, Austria)
Jonathan Bowen (Museophile Limited, UK)
Mirko Conrad (The MathWorks GmbH, Germany)
John Derrick (University of Kent, UK)
Bernd Finkbeiner (Universität des Saarlandes, Germany)
Marie-Claude Gaudel (Universite Paris-Sud, France)
Susanne Graf (Verimag, Grenoble, France)
Yuri Gurevich (Microsoft Research, USA)
Alexander S. Kossatchev (ISP RAS, Russia)
Darko Marinov (University of Illinois, USA)
Bruno Marre (Universite Paris-Sud, France)
Jefferson Offutt (George Mason University, USA)
Alexander K. Petrenko (ISP RAS, Russia)
Alexandre Petrenko (Computer Research Institute of Montreal, Canada)
Nikolai Tillmann (Microsoft Research, USA)
Jan Tretmans (Embedded Systems Institute, Eindhoven, The Netherlands)
Nina Yevtushenko (Tomsk State University, Russia)

Additional Referees

Sergiy Boroday
Bernard Botella
Klaus Dräger
Khaled El-Fakih
Nicolas Gorse
Maxim Gromov

Hesham Hallal
Lars Kuhtz
Steven Lauterburg
Chao Liu
Eda Marchetti
Vadim Mutilin

Anne Proetzsch
Rene de Vries
Nicky Williams
Marcelo d’Amorim
# Table of Contents

Can a Model Checker Generate Tests for Non-Deterministic Systems?  
*Sergiy Boroday, Alexandre Petrenko and Roland Groz*  
1

Measuring a Java Test Suite Coverage using JML Specifications  
*Frédéric Dadeau, Yves Ledru, and Lydie du Bousquet*  
17

Handling Model Changes: Regression Testing and Test-Suite Update with Model-Checkers  
*Gordon Fraser, Bernhard K. Aichernig, and Franz Wotawa*  
29

A Global Algorithm for Model-Based Test Suite Generation  
*Anders Hessel and Paul Pettersson*  
42

Generating Scenarios by Multi-Object Checking  
*Maik Kollmann and Yuen Man Hon*  
54

A Case Study in Matching Test and Proof Coverage  
*Lydie du Bousquet, Yves Ledru, Frédéric Dadeau, and Faryel Allouti*  
66

Automated Verification of Completeness and Consistency of Abstract State Machine Specifications using a SAT Solver  
*Martin Ouimet and Kristina Lundqvist*  
78

Towards the Integration of Visual and Formal Models for GUI Testing  
*Ana C. R. Paiva, João C. P. Faria, and Raul F. A. M. Vidal*  
90

Testing Planning Domains (without Model Checkers)  
*Franco Raimondi, Charles Pecheur, and Guillaume Brat*  
102
Can a Model Checker Generate Tests for Non-Deterministic Systems?

Sergiy Boroday¹, Alexandre Petrenko¹, Roland Groz²

¹ CRIM, Centre de recherche informatique de Montréal
550 Sherbrooke Str. West, Suite 100, Montreal, H3A 1B9, Canada
{Boroday, Petrenko}@crim.ca
² Institut National Polytechnique de Grenoble
F-38402 St Martin d'Hères Cedex, France
Roland.Groz@imag.fr

ABSTRACT: Modern software is increasingly concurrent, timed, distributed, and therefore, non-deterministic. While it is well known that tests can be generated as LTL or CTL model checker counterexamples, we argue that non-determinism creates difficulties that need to be resolved and propose test generation methods to overcome them. The proposed methods rely on fault modeling by mutation and use conventional (closed) and modular (open) model checkers.

Keywords: Testing, Software, Black Box, Test Generation, Verification, Model Checking, Module Checking.

1 Introduction

Test generation from deterministic specifications using verification techniques and tools is a well-known approach. It is suggested by [47], and refined by several authors, e.g., [13], [40], [8], [37], [39], [27], [21]. However, modern systems are non-deterministic due to asynchrony, concurrency, multithreading, timing issues, non-observable or non-controllable elements. Moreover, even if an implementation under test (IUT) itself can be regarded as deterministic, in a model or specification, non-determinism may occur due to incompleteness of knowledge of implementation choices, limitations of a modeling language, and abstractions. Conservative model abstraction is widely used to reduce complexity (state space) or to remove constructs, which are difficult for simulation and verification, (e.g., time aspects [1]). In declarative object oriented conceptual modeling, non-determinism allows to better reflect inherent non-determinism of the domain, reduce complexity, and achieve a better separation of concerns [4]. The problem of coping with non-determinism is a long-standing one in protocol testing [10], [17-20], [23], [38].

We present several approaches to test generation for non-deterministic specifications. We argue that a non-deterministic specification/implementation poses
certain difficulties, especially, when one targets not only weaker tests that allow for inconclusive verdicts, but also tests that deliver definitive and conclusive results no matter which system’s branch is executed. It is well known that derivation of tests for nondeterministic models is more computationally difficult or even impossible [1]. Moreover, we demonstrate in this paper that naïve approaches for coping with non-determinism could even simply fail, i.e., lead to inconsistent results. We define two different types of tests (weak and strong) which coincide in the case of deterministic specifications. A weak test is usually associated with some assumptions, e.g., if a test is executed sufficiently often, all possible reactions of a non-deterministic IUT are observed [31], [35]. Unfortunately, such assumptions do not apply to the case when non-determinism occurs in the specification due to conservative (or existential) abstraction. Thus we pay special attention to derivation of strong tests, also known as separating sequences in the context of FSM (Finite State Machine) testing [46].

Non-determinism could pose some problems even for white-box testing [26]. However, here we target the case of black-box (or functional) testing, when additional difficulties occur due to lack information of implementation details and limited observation. Only input and output variables are observable. The state and hidden (internal) variables of the IUT are not observable, so additional efforts to propagate faults into observable errors are required.

Among various test generation approaches we favor in this paper mutant-based testing, which is one of the most complex, but promising strategies to detect faults and fight the state explosion [37], [9]. Unfortunately, this technique is often associated with a so-called mutant explosion. While, traditionally, mostly deterministic mutants are considered, in [7] mutant-based test generation is extended for non-deterministic mutants. Introducing non-deterministic mutants alleviates the problem of mutant explosion and leads to a more general testing framework. Previously [39], we applied model checking to generate a confirming sequence, kind of a partial Unique Input Output sequence (UIO) [11], often used in FSM based testing and related to state identification and checking problems. Mutant-based technique could be applied to generate UIO using a mutation operator that changes the set of initial states to states which are not initial in the specification [8] (at least for specification with no equivalent states) [43].

In the context of mutation-based testing, black-box testing is also sometimes referred to as propagating faults to output. While the white-box testing (specification coverage) is often seen as a totally different testing paradigm, there exists a connection between the two types of testing. In fact, to “kill” (detect) a mutant obtained simply by mutating an output, a test that covers the affected transition (or, in the case of Moore machines or Kripke structures, a state) is necessary and at the same time sufficient. The problem of finding mutant-killing tests can be reduced to the problem of reaching states of a module composed of a specification and faulty sub-modules which satisfy a given property. This approach could be used to derive tests that combine specification coverage and fault propagation [39], [7]. However, those interested in coverage based techniques could find extensive literature elaborating usage of model checking tools for particular data or control flow coverage criteria, such as [13], [26], [42], [14].
While for finite state machines (FSM), deterministic as well as non-deterministic, test generation is a well studied theoretical problem, e.g., [22], [41], [38], [1], methods developed for classical FSM are rarely applied for real size specifications due to state explosion problem. The current trend is to use model-checking [12] technology, which embraces various sophisticated optimization techniques to cope with state explosion problem, such as BDD, partial orders, and SAT.

While previously we studied the problem of test derivation for a communicating extended FSM (CEFSM) model [7], now we cast the problem in the framework of the model checking theory which is traditionally based on Kripke structures and modules. In this paper, we generalize and further elaborate our and previously known results for model checking driven test generation for the case of non-deterministic specifications and implementations (mutants). Two types of tests, called strong and weak, are distinguished. Some properties of these tests, such as their relation to fairness are established. For the most general and difficult case of non-determinism, a novel test generation approach, based on modular model checking (module checking) of a composition of the mutant and specification is proposed. An incremental test generation approach that involves observers (transformed specifications or mutants) and traditional model checking is also outlined. Several special cases of test generation which could be resolved with traditional model checking are discussed. Counterexamples are built to demonstrate where naïve approaches to test generation with model checkers fail. The definitions and results are cast in a formal setting, based on the definition of a module which is often used in recent studies on validation of component based and modular software.

The paper is organized as follows. The next section introduces necessary definitions of test, module, model and module checking. Section 3 discusses test generation from a counterexample derived by model or module checking in the presence of non-determinism. Section 4 discusses how our results apply to the case of multiple mutants. In Section 5, we briefly discuss some related work and conclude in Section 6.

2 Preliminaries

Here we introduce the necessary notions and notations. Unlike most of FSM testing literature, which is usually based on Mealy machines [31], our work is based on the notion of a module, which could be seen as a Kripke structure with a partition of atomic propositions onto input, output, and internal (hidden) [29]. In Mealy machines, differently from Kripke structures, the labels are assigned to transitions and not to states. In some cases, Mealy machines allow for more intuitive and compact specifications than Kripke structures, especially in the black-box testing context. However, our choice is motivated by the fact that temporal logics, used in model checking, are traditionally defined over Kripke structures. As we later show, the input-output behavior of a module could be modeled by a Mealy machine. In the presence of hidden variables, the model of extended finite state machine [39] could be used to obtain a compact representation of the module.
2.1 Model Checking

A Kripke structure is a tuple \(Kr = (AP, W, R, W_0, L)\), where
- \(AP\) is the set of atomic propositions;
- \(W\) is the set of states;
- \(R \subseteq W \times W\) is the transition relation;
- \(W_0 \subseteq W\) is the set of initial states;
- \(L : W \rightarrow 2^{AP}\) is the labeling function which maps each state into a set of atomic propositions that hold in this state.

For \((w, w') \in R\), we say that \(w'\) is a successor of \(w\). We say that a Kripke structure is deadlock-free if every state \(w\) has at least one successor.

An infinite sequence of state successors is called a path. A path, starting from an initial state is called an execution path. A path is called fair \([45]\) if for each state that occurs infinitely often in the path each outgoing transition is taken infinitely often.

Usually, model checking deals only with infinite sequences. A work-around to deal with finite executions is to infinitely repeat the last state.

In this paper, atomic propositions are also seen as Boolean variables, which valuate to 1 (true) when corresponding propositions hold in the state and to 0 (false) otherwise.

Hereafter, we deal only with usual CTL and LTL syntax and semantics over deadlock-free Kripke structures \([16],[29]\).

These temporal logics extend the usual propositional logic with following temporal combinators, Finally (eventually), Globally (universally), Until, and, in case of CTL, path quantifiers All and Exists \([16]\). Beside standard conjunction and negation we use a logic equality combinator, denoted \(\varphi \leftrightarrow \psi\), or simply \(\varphi = \psi\).

Formulas, where each combinator F, G, U is immediately preceded by either quantifier A or quantifier E, constitute a temporal logic CTL, often supported by model checkers.

A model checking problem consists in checking whether a Kripke structure \(Kr\) satisfies a formula \(\varphi\) in all the initial states, denoted \(Kr \models \varphi\). A counterexample is a path of the Kripke structure, path prefix, or set of paths (in case of a complicated CTL property) that causes formula violation. Most model checkers report one or several minimum counterexamples.

2.2 Modular Specifications

A composition of two Kripke structures \(Kr = (AP, W, R, W_0, L)\) and \(Kr' = (AP', W', R', W'_0, L')\) is a Kripke structure \(Kr \parallel Kr' = (AP'', W'', R'', W''_0, L'')\), where
- \(AP'' = AP \cup AP'\)
- \(W'' = \{(w, w') : L(w) \cap AP' = L'(w') \cap AP\}\)
- \(R'' =\{(w, w'), (s, s') : (w, s) \in R, (w', s') \in R'\} \cap W''\)
- \(W''_0 = (W_0 \times W'_0) \cap W''\)
- \(L''(w, w') = L(w) \cup L(w')\) for \((w, w') \in W''\).
Thus, the composition synchronizes over state labels shared by the components. In other words, each state of the composition is composed of state of the first component and state of the second component, such that each atomic proposition that belongs to both Kripke structures is present or absent in both states simultaneously.

A module is a triple \((Kr, I, O)\), where \(Kr = (AP, W, R, W_0, L)\) is a Kripke structure, and \(I, O \subseteq AP\) are disjoint sets of input and output variables. \(H = AP \setminus (I \cup O)\) is called the set of hidden (internal) variables. While hidden variables may appear redundant, we need them here for technical reasons. A module is closed, if \(I = \emptyset\), otherwise it is open. Intuitively, in every state \(w\), the module reads \(L(w) \cap I\), stores internally \(L(w) \cap H\), and outputs \(L(w) \cap O\). \(Inp(w) = L(w) \cap I\) is called the input of the module in the state \(w\). \(Out(w) = L(w) \cap O\) is called the output of the module in the state \(w\).

A module is called deterministic if for each state \(w\) and each \(i \subseteq I, w\) has at most one successor state \(s\) with the input \(Inp(s) = i\), moreover, for all \(w, s \in W_0\), \(Inp(s) = Inp(s)\) implies \(w = s\). A module is non-deterministic, otherwise.

Given a sequence \(w_1, w_2, \ldots, w_k\) of successor states of a module \((Kr, I, O)\), starting from an initial state, we say that \((Kr, I, O)\) produces an output sequence \(Out(w_1)Out(w_2) \ldots Out(w_k)\) in response to an input sequence \(Inp(w_1)Inp(w_2) \ldots Inp(w_k)\), while \(Inp(w_1)Out(w_1)Inp(w_2)Out(w_2) \ldots Inp(w_k)Out(w_k)\) is called an input-output sequence.

Similar to Mealy machines, a module \((Kr, I, O)\) is called observable [46], if the module \((Kr, I \cup O, \emptyset)\), obtained from the initial module by moving all the output variables into the input variable set, is deterministic. Otherwise, the module \((Kr, I, O)\) is called non-observable. Intuitively, observable non-determinism is a simple form of non-determinism, when a path, taken by the module, could be deduced from the observed input–output sequence. Non-observable non-determinism is harder to deal with. Fortunately, for each non-observable module, an observable module with the same set of input-output sequences exists. Such module can be constructed by a well known powerset construction procedure.

Input \(i\) is enabled in the state \(w\) if \(w\) has at least one successor state \(s\) with input \(Inp(s) = i\). Otherwise, input is disabled. A module is input enabled (completely defined) if each input labels an initial state and is enabled in every state. In this paper, we consider only input-enabled specification and mutant modules, and, hence, deadlock-free.

A module, which constitutes our running example of a specification is shown in Fig.1a. For simplicity, values of the input variable \(i\) and the output variable \(o\) are depicted as pairs of Boolean values, where 0 denotes false value, and 1 denotes the true value. Each state has a unique label. The specification has two states, \(w_1\) labeled with 0/0, and \(w_2\) labeled with 1/1, both states are initial. The specification can also be represented as a Mealy machine (Fig. 1d). The transition relations of all the modules are defined in such way that states, where \(i = o = 0\) or \(i = o = 1\), and only them, are successors of all the other states. All three modules on Fig. 1 are input enabled.
A composition of the modules $M = (K^M, I^M, O^M)$ and $S = (K^S, I^S, O^S)$, such that no hidden variable of one module is a variable of another ($AP^M \cap H^S = AP^S \cap H^M = \emptyset$), and sets of output variables are disjoint ($O^S \cap O^M = \emptyset$), is $M || S = (K^M || K^S, (I^M \cup I^S) \setminus (O^S \cup O^M), (O^S \cup O^M))$. If needed, output and hidden variables are renamed for the composition. Note that our definition allows common inputs in the module composition.

A union of the modules $M = (K^M, I^M, O^M)$, where $K^M = (AP^M, W^M, R^M, W^M_0, L^M)$ and $S = (K^S, I^S, O^S)$, where $K^S = (AP^S, W^S, R^S, W^S_0, L^S)$ with mutually exclusive state sets is the module $M \cup S = ((AP^M \cup AP^S, W^M \cup W^S, R^M \cup R^S, W^M_0 \cup W^S_0, L), I^S \cup I^M, O^S \cup O^M)$, where $L(w) = L^M(w)$ if $w \in W^M$, and $L(w) = L^S(w)$ if $w \in W^S$. If the state sets intersect, states could be renamed. One can easily see that the set of input-output sequences of the union is the union of sets of input-output sequences of the original modules.

Model checking of specifications with infinite state space is still limited to small to medium specification and undecidable in the most general case. Thus, hereafter, we consider only modules with finite number of states, variables, and propositions.

2.3 Module Checking

A module satisfies $\varphi$, if the Kripke structure of the module satisfies $\varphi$. However, a formula (property) that holds on a module in isolation does not always hold when the module is embedded into a system of communicating modules. Often it is important that a property is satisfied when the module reactively communicates with other (possibly unknown) modules. To address this problem, a new technique, more general than conventional model checking, is introduced [29].

A module $M$ reactively satisfies $\varphi$, denoted $M \models_r \varphi$, if $E \parallel M \models \varphi$ for all deadlock-free modules $E \parallel M$, where $E$ is a module such that no hidden variables of one module is a variable of another module, and no output variable of one module is an output variable of another module. The module $E$ represents a possible environment that does not block the module $M$. Checking reactive satisfaction of a formula

Fig. 1. a) The specification; b) and c) mutants; d) Mealy machine.
constitutes the problem of modular model checking with incomplete information or module checking, for short [29]. For LTL, module checking coincides with model checking.

2.4 Testing

We consider here mutation-based test generation, where faults are modeled using the same formalism as a specification, and mutants are obtained from the specification by applying mutation operations. While mutation is traditionally performed on code, recently, a range of specification mutation operators is suggested [9], [37].

Let $S$ be a module that models a specification and $M$ be a module that models a faulty implementation (mutant) that share the same set of input and output variables.

A finite sequence of inputs $i_1 \ldots i_k$, where $i_j \subseteq I$, is called a weak test for $S$ and $M$ if in response to it $M$ produces an output sequence that $S$ cannot.

A finite sequence of inputs $i_1 \ldots i_k$ is called a strong test for $S$ and $M$ if any output sequence $M$ produces in response to $i_1 \ldots i_k$ cannot be produced by $S$.

While, in certain cases, it might be more efficient to use a test suite or adaptive (on-the-fly) tests instead of simple tests defined above, for simplicity, in this work, we target only single preset tests.

If $S$ and $M$ are deterministic, the notions of strong and weak tests coincide; in the non-deterministic case, a weak test may exist even when there is no strong test.

**Weak Tests and Fairness.** Here we show that, under fairness assumption, repeatedly executed (a sufficient number of times) weak test reveals the fault. In order to be able to repeat a test, a reliable reset to set the module into all its initial states is needed. Such a reliable reset could be modeled as follows. Let $S_{reset}$ and $M_{reset}$ be modules obtained from the original modules by adding a reset input variable to both $S$ and $M$. Each original initial state is replaced by a reset state with the same variables along with the designated variable reset and the same set of successors. Each reset state is a successor of all states. Let $(i_1 \ldots i_k)^\omega$ be a module, which infinitely often repeats $i_1 i_2 \ldots i_k$ as its output (which, in our case, become inputs for other modules). The following proposition holds.

**Proposition 1.** If $i_1 \ldots i_k$ is a weak test for $S$ and $M$, then each fair execution path of $M_{reset} \parallel (i_1 \ldots i_k(\text{reset}))^\omega$ and each fair execution path of $S_{reset} \parallel (i_1 \ldots i_k(\text{reset}))^\omega$ contain different output sequences.

Obviously, no weak test exists, if and only if each input-output sequence of the mutant is also an input-output sequence of the specification. In case of a deterministic specification, a weak test does not exist if and only if the specification and mutant have the same set of input-output sequences.

**Strong Tests.** If there exists a strong test for modules $S$ and $M$, these two modules are called separable (similar to non-deterministic FSM, see, e.g., [46]). In case of a deterministic specification, a strong test does not exist if and only if each input-output sequence of the specification is also an input-output sequence of the faulty module.
A strong test is symmetric: each strong test for a pair \((S, M)\) is also a strong test for \((M, S)\), i.e., it is not important which of the two is the specification or the mutant.

**Complexity Issues and Justification for Modular Model Checking.** Derivation of strong tests and its complexity is discussed by Alur et al [1], where strong tests for two modules are referred as preset distinguishing strategies for two machines. A preset distinguishing strategy is seen as a winning strategy for a (blindfold) \(\exists\forall\) game with incomplete information. Since such games are PSpace complete, the same is true for the preset distinguishing strategy existence problem, and the length of a preset distinguishing sequence is exponential (in the worst case). The proposed method for the preset distinguishing strategy generation is based on derivation of a special type of a power set module of exponential size that resembles Gill’s successor tree [22]. Thus, such an automaton could be constructed and fed to a model checker. However, model checking technology still may face difficulties with large space state and specification size. Thus, we try to build a smaller specification, applying a game-theoretic model checking known as module checking.

At the same time, we discuss possible use of more traditional and widespread LTL or CTL model checking techniques for simpler cases of the most general problem. Moreover, we show that candidate test verification and, hence, incremental test derivation is possible for non-deterministic specifications and mutants. Yet, we doubt that one could use conventional LTL or (non-modular) CTL model checking technique to derive strong test or decide its existence without significant computational efforts on the model transformation. An exponential growth of the size of the system or property being model checked is expected. Indeed, the complexity of CTL model checking is linear in terms of state space, while finding strong tests is PSpace-complete.

# 3 Mutant-Based Testing by Model and Module Checking

Test generation for a deterministic specification and deterministic mutant modules is well understood. However, we first cast this problem in our framework to generalize it later for non-deterministic cases.

## 3.1 Deterministic Case

For the deterministic case we define a system that consists of a specification and mutant modules. Then we model check a formula that states equality of the outputs of specification and mutant modules, so a counterexample will give a test. In our framework, this simple idea could be formalized as follows.

In order to compose modules with common hidden and output variables \(O \cup H\), we introduce a renaming operator. Formally, the renaming operator ‘ is defined on hidden and output variables of a module: \((p)’ = p’, where \(p’\) is not in the set of the atomic propositions of these modules. We lift the operator to sets of variables and
modules. Also, let \( p = p' \) hold in a state when both atomic propositions \( p \) and \( p' \) are simultaneously hold or do not hold in this state.

**Theorem 1.** Let \( S = (Kr^S, I, O) \) and \( M = (Kr^M, I, O) \) be two deterministic modules with the same input and output variables.

\[
S \parallel M' \models AG \bigwedge_{p \in O} (p = p')
\]

iff there is no (strong or weak) test.

\( \bigwedge_{p \in O} (p = p') \) is a shorthand for \( o_1 = o'_1 \land o_2 = o'_2 \land \ldots \land o_{|O|} = o'_{|O|} \), which formally denotes the equality of outputs of \( S \) and \( M \). (While a more elegant formula, \( Out' = (Out) \) could be used instead; some readers may find it confusing.)

The idea of the proof is as follows. The set of output variables of the composition \( S \parallel M' \) is \( O \cup O' \). The composition is synchronized only by input in the sense that for each pair of execution paths of modules \( S \) and \( M' \), that share the same sequence of inputs, pairs of states form an execution path of the composition. Each execution path of composition is a sequence of pair of states of execution paths of \( S \) and \( M' \), that correspond to the same input sequences. The logic formula simply states that each output variable \( o \in O \) always valuates as the corresponding renamed variable \( o' \) on each path of the composition. That is the output of \( M \) coincides with the output of \( S \) for each input sequence, which is a necessary and sufficient condition for the absence of strong test for two finite modules.

Note that formula in Theorem 1 belongs both to CTL and LTL. Since LTL module checking coincides with model checking, \( S \parallel M' \models AG \bigwedge_{p \in O} (p = p') \) is equivalent to \( S \parallel M' \models _r AG \bigwedge_{p \in O} (p = p') \).

An input sequence defined by a (finite) counterexample to the expression in Theorem 1 constitutes a strong test case. This idea is known and widely exploited. In fact, the set of such input sequences, defined by counterexamples, and the set of strong tests coincide.

**Example.** To illustrate Theorem 1, we consider the modules shown in Fig. 1a and Fig. 1b. While the specification always reacts to input \( i = 1 \) with output \( o = 1 \), the mutant starts with output \( o = 0 \). Thus, \( S \parallel M' \models G (o = o') \) does not hold. \( i = 1 \) is both the shortest counterexample and the shortest test.

### 3.2 Non-Deterministic Mutant

**Weak Test Derivation.** Even if a specification is deterministic, a mutant may still exhibit some non-determinism, e.g., related to data races or abstraction. This is why it is interesting to consider the case of a deterministic specification and non-deterministic mutants.

**Theorem 2.** Let \( S = (Kr^S, I, O) \) and \( M = (Kr^M, I, O) \) be two modules with the same input and output variables, where \( S \) is deterministic, while \( M \) is possibly non-deterministic.

\[
S \parallel M' \models AG \bigwedge_{p \in O} (p = p')
\]

iff there is no weak test for \( S \) and \( M' \).
As we noted before, $S \parallel M' = AG \bigwedge_{p \in O} (p = p')$ is equivalent to $S \parallel M' = r \bigwedge_{p \in O} (p = p')$.

An input sequence defined by a finite counterexample to the above formula could serve as a weak test.

**Example.** Consider the specification in Fig. 2a and the mutant as in Fig. 2b, but with all three states being initial. $i = 1$ is a weak test, since the mutant can produce the output $o = 0$, which the specification cannot produce in response to the input $i = 1$. At the same time, the mutant can produce output $o = 1$, as the specification.

Determinism of the specification is essential. In the case when both modules are non-deterministic, the formula $G \bigwedge_{p \in O} (p = p')$ could be violated even when no strong or weak test exists. For example, let both, specification and faulty, modules be as in Fig. 2b, but with all states being initial. Thus, both modules are non-deterministic and coincide. The formula of Theorem 2 does not hold, since there exists an execution path, such that $o \neq o'$ already in the first state ($w_2, w_3$), though no weak test exists.

**Strong Test Generation.** In order to determine a strong test, we first replace a non-observable mutant by an observable one with the same set of input-output sequences. The following proposition states that this transformation does not affect test existence.

**Proposition 2.** If $M_1$ and $M_2$ have the same set of input-output sequences then each strong test for $S$ and $M_1$ is also a strong test for $S$ and $M_2$.

The idea behind strong test derivation relies on transforming an observable mutant module into an observer [24]. We build a module $\text{Obs}(M)$ for a given module $M$ by the following sequence of transformations. Each output of the original module becomes an input of the new module. A hidden variable $\text{found}$ is defined, thus, if present, original hidden variables are removed. If needed, determinization is performed by powerset construction. In all non-trivial cases, the obtained module is not input-enabled due to inputs, obtained from the output set $O^M$ of the original module $M$. The module is completed to an input-enabled module with a set of additional “sink” states. The variable $\text{found}$ valuates true only in these states. More formally, for each state and each disabled input $i$, a successor “sink” state, labeled with $i \cup \{\text{found}\}$ is added. For each input $i$ that does not label an initial state, an additional initial state labeled with $i \cup \{\text{found}\}$ is defined. Each of the added states is a successor of all the other added states.

Note that determinization is required only for non-observable module $M$; otherwise, the obtained module is deterministic due to performed inversion of outputs into inputs. Possibly, approximate conservative determinization [44] could be applied.

The following holds for the obtained modules.

**Theorem 3.** Let $S = (K^S, I, O)$ and $M = (K^M, I, O)$ be two modules with the same input and output variables, where $S$ is deterministic, while $M$ is possibly non-deterministic.

\[ S \parallel \text{Obs}(M) = AG \text{ not found} \]

iff there is no strong test.
Note that, since LTL module checking coincides with model checking, 
\( S \parallel \text{Obs}(M) \models \text{G not found} \) is equivalent to \( S \parallel \text{Obs}(M) \models \text{G not found} \).

### 3.3 Non-Deterministic Modules

**Weak Test.** A weak test can also be generated using a composition of a module with an observer. However, the observer should be built from the specification, rather than from the mutant.

**Theorem 4.** Let \( S = (K^S, I, O) \) and \( M = (K^M, I, O) \) be two modules with the same input and output variables.

\[
M \parallel \text{Obs}(S) \models \text{AG not found}
\]

iff there is no weak test case for \( S \) and \( M \).

Note that, since LTL module checking coincides with model checking, 
\( M \parallel \text{Obs}(S) \models \text{AG not found} \) is equivalent to \( M \parallel \text{Obs}(S) \models \text{G not found} \).

**Strong Test Verification.** The case of strong test derivation when both specification and mutants are non-deterministic is the most complex among those considered here, one may use approximation or ad-hoc methods to solve it. In this case, a verification procedure may be needed. Thus, we discuss how to check whether a candidate input sequence constitutes a strong test.

In order to model check whether a given candidate input sequence is a test, we define a tester module, called Tester, which simply feeds a given sequence to the specification and mutants. Tester(\( \alpha \)) for a given input sequence \( \alpha = i_1...i_k \), is a module with the empty set of input variables, a hidden variable (flag) \( h \), the set of states \( \{w_0, w_1, ..., w_k\} \), and transition relation \( \{(w_j, w_{j+1}) : 0 \leq j \leq k - 1\} \cup \{(w_k, w_k)\} \), the labeling function \( L \) such that \( L(w_j) = i_{j+1} \) for \( 0 \leq j \leq k - 1 \), and \( L(w_k) = \{h\} \). The loop \( (w_k, w_k) \) and a flag \( h \) are needed, because in our framework, inspired from [29] model checking (as well as module checking) is defined only on deadlock-free modules.

**Theorem 5.** Let \( S = (K^S, I, O) \) and \( M = (K^M, I, O) \) be two modules with the same input and output variables and \( \alpha \) be an input sequence.

\[
S \parallel M' \parallel \text{Tester}(\alpha) \models \text{AF not (h} \lor \bigwedge \text{p} \in O(p = p'))
\]

iff the input sequence \( \alpha \) is a strong test.

The idea of the proof is as follows. The formula formalizes the intuition that a test is strong if it guarantees that in the course of test execution, a mutant, sooner or later (eventually), produces an output, different from any output that the specification is capable of.

Note that \( S \parallel M' \parallel \text{Tester}(\alpha) \models \text{F not (h} \lor \bigwedge \text{p} \in O(p = p')) \) is equivalent to \( S \parallel M' \parallel \text{Tester}(\alpha) \models \text{F not (h} \lor \bigwedge \text{p} \in O(p = p')) \).

For most of “complete” model-checking algorithms and tools, which compose modules prior to property analysis, replacing \( M' \) by \( \text{Obs}(M) \) may somewhat reduce space state, but the gain is relatively small for on-the-fly model checking, when only a fragment of the composition, which is relevant to the property, is usually constructed.
An incremental generation of a strong test can be performed by consecutive verification of all candidates of a given length. If the state number of modules is finite, an upper bound of the test is known [1]. Technically, it is possible to define a module that consecutively tries all possible test candidates of the given length. Such approach could be faster than model checking multiple systems, but we do not see how it could be organized efficiently in the terms of the memory consumption.

**Strong Test Derivation by Module Checking.** To derive a strong test for the most general case, when both specification and mutant are non-deterministic, we introduce two auxiliary operators. One operator \( \text{HideOut}(Kr, I, O) = (Kr, I, \emptyset) \) is a blindfold operator. Intuitively, the blindfold operator preserves the structure and inputs of the original module, but hides output variables from the environment, by placing them into the set of hidden variables of the resulting module. Another additional operator\(^1\) \( \text{AddIn} \), which adds a new, single initial state \( w_0 \), such that \( L(w_0) = \emptyset \), and transitions from \( w_0 \) lead to all the former initial states (and only to them).

**Theorem 6.** Let \( S = (Kr^S, I, O) \) and \( M = (Kr^M, I, O) \) be two modules with the same input and output variables.

\[
\text{HideOut(AddIn}(\langle S \parallel M') \rangle), \text{ EG } (\bigwedge_{p \in O}(p = p'))
\]

iff there is no strong test for \( S \) and \( M \).

The intuition behind this statement is that two finite modules produce at least one common output sequence for any input sequence if and only if there is no strong test for these two modules. Input sequences are produced by a module checking environments. The condition of a blind environment is needed since here we are not interested in adaptive testers and testers that could prevent (block) certain outputs. Blindness of environment ensures that the formula is reactively satisfied if and only if it is satisfied for all the possible environments each state of which has a single successor. Thus, multiple tests are not addressed. Therefore, the reactive satisfaction of the formula of Theorem 6 on the blindfolded composition of the specification with the mutant formalizes the above necessary and sufficient condition of strong test non-existence. Since module checking allows for both, output sensitive (adaptive) and blind, environments, we hide the outputs of the composition with a designated operator \( \text{HideOut} \). Note that \( O \) in the formula does not refer to the set of output variables of \( \text{HideOut(AddIn}(\langle S \parallel M') \rangle) \). \( O \) denotes the set of the output variables of the original module \( S = (Kr^S, I, O) \). In \( \text{HideOut(AddIn}(\langle S \parallel M') \rangle) \) these variables are hidden.

While, unlike the previous cases, a minimum counterexample may involve several execution paths, it will provide a unique finite input sequence. To obtain a strong test, though, one should disregard the first (empty) input of this sequence.

Note, since the formula uses path existence quantification, it does not belong to any universal logic. In this case, replacing reactive satisfaction by usual would change the meaning of formula. In the context of the conventional model checking, formula of Theorem 6 would state that each sufficiently long sequence of inputs constitutes a strong test for given \( S \) and \( M \). Possibly, in (conventional) model

---

\(^1\) This operator is needed only for non-deterministic specifications or mutants, which have several different initial states that share same input.
checking setting, a specific for $M$ (but still independent from $S$) non-separability condition could be expressed in the form of the so-called tree automaton [28], but we doubt that this is always possible in CTL.

4 Multiple Mutants

Here we discuss derivation of a test which targets several mutants. The problem could be reduced to the case of a single mutant by merging a set of mutants $\{M_1, \ldots, M_l\}$ into a single module, $M_1 \cup \ldots \cup M_l$.

**Proposition 3.** An input sequence is a strong test for $S$ and the module $M_1 \cup \ldots \cup M_l$ iff it is a strong test for $S$ and each of the modules $M_1, \ldots, M_l$.

The proposition does not hold for weak tests.

Such transformation of several mutants into “meta-mutant” should be consider with care, since a test for the specification and the “meta-mutant” may not exist even if there exists a single test for each mutant. (The problem persists for multiple tests also). Consider the following example.

**Example.** Consider the specification in Fig. 1a, and mutants in Fig. 1b and Fig. 1c. The first mutant has an additional state $w_3$ labeled with 1/0, the initial states are $w_1$ and $w_3$. The second mutant has an additional state $w_4$ labeled with 0/1, the initial states are $w_2$ and $w_4$. Any strong test for the specification and the first mutant, should start with $i = 1$. Similarly, any strong test for the specification and the second mutant should start with $i = 0$. Thus, there is no single test for the specification and both mutants at the same time.

A finer approach is to build a composition of the specification with all the mutants. However, outputs of each of them should be renamed. Unlike the case of the merged modules, for such a multi-component composition, the testing remains tractable even if there is no input sequence that is a strong (weak) test with respect to all the mutants. For example, the problem of deriving a test which is strong for a maximal number of mutants could be considered, similar to a confirming sequence, a kind of partial UIO, generation [40]. Such an optimization problem may look alien to classical model checking, but many model checking tools, such as ObjectGeode [36], provide means to collect statistics. Possibly, certain metrics, which assign a weight for each mutant, could be implemented.

5 Related Work

As far as we know, reuse of verification techniques for test generation is first suggested in [47], though, in our opinion, treatment on non-determinism is rather rudimentary in this work. In order to cope with non-determinism in a specification, [37] suggests synchronizing “non-deterministic choices”, i.e., in value of variables that are updated in a non-deterministic fashion (for the case when each state is uniquely defined by the values of variables). In terms of SMV language, it consists in
declaring a variable global and removing update operations from a mutant. The approach, as suggested, works in simple cases. However, it is not clear how it applies to the most general case. For example, we consider the case when one hidden variable is updated in a non-deterministic fashion, but its value is revealed via an output variable only in a next step. The approach of [37] could result in the false tests for mutations that negate the output value in that step. Our observer based test generation could be seen as a more general version of the method sketched in [37].

Some obstacles in test generation for tricky coverage criteria from CTL model checking counterexample are resolved by adding an additional logic operator [26]. In this paper, instead of applying a rare temporal logic, we cope with non-determinism using module checking.

6 Conclusion

We discussed difficulties in test generation for non-deterministic systems not only using complexity-theoretical arguments, but also by demonstrating how naïve approaches may fail and proposed several solutions, which allow one to use a modular or conventional CTL (and, sometimes, LTL) model checker to generate tests automatically. As usually, counterexamples, generated by a model checker, could be used to build tests. We demonstrated that in the most general case of non-deterministic specifications and implementations, the existence of a test could be verified using the module checking approach. Alternatively, an incremental approach, where each candidate test is verified using a conventional model checking is proposed. While the incremental approach may appear inefficient, our preliminary experiments [25] give encouraging results. Moreover, incremental approach could rely on a larger range of tools and makes possible the use of efficient techniques of bounded model checking [6], when the behavior of the specification is analyzed only on executions of a given length.

Our future plans include test derivation experiments with module checking and contingent AI planning. In view of a currently rather limited choice of module checking tools, we hope that our ideas of using module checking for test derivation could motivate development of new algorithms and tools.

References

Measuring a Java Test Suite Coverage using JML Specifications*

Frédéric Dadeau, Yves Ledru, and Lydie du Bousquet

Laboratoire LSR-IMAG
BP. 72
38402 Saint Martin D'Hères cedex
Email: {Frederic.Dadeau,Yves.Ledru,Lydie.du-Bousquet}@imag.fr

Abstract. We propose in this paper a way to measure the coverage of a Java test suite by considering the JML specification associated to the Java program under test. This approach is based on extracting a predicate-based graph from the JML method specifications. We then measure the coverage of this latter w.r.t. nodes of the graph that are visited by the test suite. In addition, we propose to check whether the test suite satisfies classical condition coverage criteria. We also introduce a tool, to be used as precompiler for Java, that is in charge of measuring and reporting the coverage according to these criteria.

Keywords: Specification coverage, test suite, Java, JML, condition coverage.

1 Introduction

The essence of testing consists in executing the system under test in order to find bugs [21]. Nevertheless, testing can not be a complete approach since exhaustive testing is not applicable; the validation engineer is often left with a test suite that did not detect any bug in the program. How can he/she be sure that the test suite that was run is relevant enough to be confident in the program? One solution is to evaluate the quality of the test suite.

Several works on test suite evaluation exist, such as exercising the test suite on mutations of a program. The most relevant technique is to measure the coverage of the test suite. Usually, the coverage is measured on the control-flow graph of the program [21], or on the data-flow of the program [24]. In addition, a specification coverage can possibly be measured.

The recent rise of annotation languages makes it possible to specify the behavior of programs (i.e. of methods) inside the source code in terms of pre- and postconditions. It provides another “vision” of what a method should do, which can also be seen as expressing low-level requirements. It also provides a black-box view of what a method should do, expressed in terms of a contract [20].

* This work has been partially funded by the RNTL POSE project, supported by the French Ministry of Research and New Technologies.
Model-based testing [2] consists in computing test suites from a model of the considered program or system. Model-based conformance testing consists in ensuring that the program does not have an unintended behavior w.r.t. its specification. This conformance can be observed through observation points or using a run-time assertion checking mechanism if the proximity of both the specification and the program makes it possible. In this context, the Java Modeling Language [16] (JML) has been introduced to act as a behavioral interface specification language (BISL) for Java programs. JML can be used as an oracle for testing, considering that if no JML assertion is ever violated during the program execution, then the test succeeds, otherwise, it fails.

A previous work [3] has introduced the principles of model-based testing from JML specifications. Thus, some JML-based coverage criteria were used to guide the test target definition. We came to the idea that the coverage criteria defined in the latter work could be used to evaluate test suites that would have been produced by several tools, using different approaches, such as combinatorial testing (e.g. JMLUnit [6], TOBIAS [18]) or random testing (e.g. JARTEGE [23]).

We propose an approach for evaluating test suites for Java programs w.r.t. the coverage of an associated JML specification, expressing the behavior and/or the requirements of the methods. In addition, we propose to check different condition coverage criteria, that are contained within the disjunctions of the predicates of the specification.

The paper is organized as follows. Section 2 introduces the modeling possibilities provided by the Java Modeling Language. The first coverage criterion, based on the method specifications, is presented in Sect. 3. Section 4 is dedicated to the condition coverage definition. The principles of the measure and especially the implementation and the experiments are detailed in Sect. 5. Section 6 presents the related work, before concluding and providing a glimpse of the future work in Sect. 7.

2 Java Modeling Language

The Java Modeling Language -JML- has been designed by Gary T. Leavens et al. at Iowa State University [16,17]. The modeling elements are displayed as annotations, embedded within the Java source code. JML is based on the design by contract principle (DBC) [20] which states that, at the invocation of a method the system has to fulfill a contract (by satisfying the method’s precondition) and as a counterpart, the method is expected to fulfill its contract (by establishing its postcondition).

JML makes it possible to express the static part of the system, such as invariants, and the dynamic part of the system, using postconditions and history constraints. The model is expressed through several clauses, identified by a keyword, and followed by a predicate. The invariant and constraint clauses respectively designate the invariant and the history constraints that apply to a class. The general behaviors of the methods are specified through method specifications, which contain specification blocks, separated by the also keyword. Each
public class Demoney {
    static final SET_MAX_DEBIT = 1;
    static final SET_MAX_BAL = 2;
    //@ invariant balance > 0 &&
    //@ balance == maxBal;
    int balance, maxBal, maxDebit;
    boolean personalized;
    /* behavior */
    // requires personalized = false;
    @
    // requires pl == SET_MAX_DEBIT;
    // requires data >= 0;
    // assignable maxDebit, maxBal;
    // ensures maxDebit = data &&
    // maxBal = old(maxBal);
    // also
    // requires pl == SET_MAX_BAL;
    // requires data == balance;
    // assignable maxDebit, maxBal;
    // ensures maxBal = data &&
    // maxDebit = old(maxDebit);
    // also
    // requires personalized = true !!
    // (pl != SET_MAX_DEBIT !!
    // data < balance) &&
    // (pl != SET_MAX_BAL !!
    // data < 0));
    // assignable maxDebit, maxBal;
    // ensures false;
    // signals (CardException e)
    // maxDebit = old(maxDebit)
    // && maxBal = old(maxBal);
    //
    public void PUT_DATA(byte pl, short data) {
        ...
    }
}

Fig. 1. An example of a JML specification

specification block displays preconditions (requires clause), normal postconditions —established when the method terminates without throwing an exception— (ensures clause), or exceptional postconditions —established when the method terminates by throwing an exception— (signals clause). The assignable clause gives the frame of the method. The JML predicate syntax is similar to the Java predicate syntax enriched with special keywords, prefixed by \, notably introducing quantifiers (forall, \exists).

An example of a JML specification is displayed in Fig. 1. This specification presents a simplified version of the Demoney electronic purse [19]. Attributes balance, maxBal, and maxDebit respectively represent the amount of money on the purse, the maximal amount of money, and the maximal debit authorized. Finally, attribute personalized states whether or not the purse has been configured, by defining the values of the latter two attributes. The method displayed here makes it possible to configure the purse, by setting either the maximal value of balance, or the maximal debit. The method may throw an exception either if the parameter pl is wrong of if the card is already personalized.

The JML Runtime Assertion Checker (RAC) [5] has been developed to check the JML specification clauses when running the program. This tool, provided in the JML releases, acts as a precompiler which modifies the source of the program to add the following verifications on the JML model: (i) checking of the preconditions and the invariant when a method is entered, (ii) catching exceptions that may be thrown and checking of the exceptional postcondition related to the considered exception before throwing the exception again, (iii) checking of the normal postconditions if the method terminates normally. Notice that the invariant and history constraints are also checked during steps (ii) and (iii).
The next section introduces our proposal to decompose the JML method specifications into behaviors, whose coverage are the measuring unit of our approach. In addition, we will consider condition coverage criteria that will add more granularity to the measure.

3 Coverage of the Method Specifications

The method specifications describe behaviors of the Java methods. A behavior is either normal if the method terminates normally (without throwing an exception) or exceptional if the method terminates by throwing an exception. Our technique is to extract a predicate-based graph from the method specifications, that gives a representation of the behaviors of the method. Traversing the graph is equivalent to create a conjunction of the label predicates on its edges.

We represent each JML method specification by a graph, as shown in Fig. 2. In this figure, $P_k(k \in 1..N)$ are the precondition predicates, $A$ gives the frame condition, $Q_k(k \in 1..N)$ are the normal postconditions, $S_p(p \in 1..M)$ are the exceptional postconditions related to the exceptions $E_p$. The terminations are distinguished by $T$, which might be either no exception indicating a normal behavior, or any of the declared exceptions $E_p$. Invariants and history constraints are (currently) not considered.

The extraction of the graph is done as follows. A first branch containing the normal behavior is built. According to the semantics of JML, when the precondition of a method specification block is satisfied then the normal postcondition

```java
/** behavior */
@ requires $P_1$;
@ assignable $A$;
@ ensures $Q_1$;
@ signals $(E_{11}$ e1$)$ $S_{11}$;
@ ...
@ signals $(E_{1M}$ eM$)$ $S_{1M}$;
@ also
@ ...
@ also
@ requires $P_N$;
@ assignable $A$;
@ ensures $Q_N$;
@ signals $(E_{N1}$ e1$)$ $S_{N1}$;
@ ...
@ signals $(E_{NP}$ eNP$)$ $S_{NP}$;
@*/

Type meth(T, p1, ...) throws E_{11},...,E_{NP} { ... }

Fig. 2. Extraction of the behaviors from a JML method specification
has to be established. Otherwise, if the precondition is not satisfied, anything may happen. This desugaring [25] can be expressed by Fig. 3. As a consequence, conditional branchings are created, in order to re-create the implication. One branching is done for each method specification block. In case of exceptional termination, if an exception is thrown, then, depending on the precondition, an exceptional postcondition has to be established. An example of such a graph is given in Fig. 4. Notice that frame conditions are not considered in the graph.

The coverage of the method specifications is achieved by covering the method specification graph. Since this graph is directed and acyclic, we do not have to cover loops. Thus, the following options can be proposed.

- **all nodes.** Achieved when a test suite activates all the nodes of the graph.
- **all edges.** Achieved when a test suite activates all the edges of the graph.
- **all paths.** Achieved when a test suite activates all the paths of the graph going from node 1 to node 0.

The hierarchy between these options is the following:

\[
\text{all nodes} \subseteq \text{all edges} \subseteq \text{all paths}
\]

We assume that only consistent paths are computed/measured. This is ensured either by writing a comprehensive JML method specification, or by using a dedicated tool, such as JML-TESTING-TOOLS constraint solving engine [4] or a theorem prover, such as Simplify [9] or haRvey [8] to prune the inconsistent paths in the method specification graph.

**Example 1 (Extraction of a Graph from a Method Specification).** Consider the method specification example given in Fig. 1. The corresponding graph is given in Fig. 4. A path is read as a conjunction of the statements appearing on its edges. On this example, path \[1 \rightarrow 9 \rightarrow 10 \rightarrow 0\] is equivalent to \(X \land T = \text{CardException} \land \text{maxDebit} = \text{old(maxDebit)} \land \text{maxBal} = \text{old(maxBal)}.\) This graph presents several inconsistent paths, among all the possible paths leading from node 1 to node 0. On the example, only paths \[1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 7 \rightarrow 0\], \[1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 0\], and \[1 \rightarrow 9 \rightarrow 10 \rightarrow 0\] are consistent.

Figure 3 illustrates the desugaring of JML method specifications, rewriting several blocks into a single one. The graph construction is based on the hypothesis that specifications are divided, as much as possible, into several blocks (left
Fig. 4. Graph extracted from the PUT_DATA method specification

part of the figure). Nevertheless, most of the JML specification writers are not used to split the specification like that and only write one huge postcondition into which case-based postconditions are guarded (middle part of the figure). Thus, a graph for such a method specification would only consider one precondition and one postcondition. Measuring the coverage of this kind of graph is not really relevant. That is why, in addition to the coverage of the predicate graph, we also consider condition coverage criteria, for the predicates of the graph. If the method specification is divided into blocks, then this additional coverage increases the granularity of our measure.

4 Condition Coverage

Condition coverage is achieved by rewriting the disjunctions embedded within the predicates on the edges composing a path. We distinguish 4 rewritings, each one of them representing a specific condition coverage criterion. These rewritings and their associated coverage criteria are given by Table 1.

Rewriting 1 consists in checking the disjunction without any changes. This is the most basic way to verify a disjunction, by choosing the first positive literal.

Rewriting 2 consists in considering each literal independently. This rewriting satisfies the Condition Coverage criterion (CC).

Rewriting 3 considers each literal in an exclusive manner, by evaluating each literal and the negation of the others. Thus, this rewriting satisfies the Full Predicate Coverage criterion (FPC) [22].
<table>
<thead>
<tr>
<th>Rewriting</th>
<th>Set of predicates to evaluate for $P_1 \lor P_2$</th>
<th>Coverage Criteria</th>
</tr>
</thead>
<tbody>
<tr>
<td>RW1</td>
<td>${P_1 \lor P_2}$</td>
<td></td>
</tr>
<tr>
<td>RW2</td>
<td>${P_1, P_2}$</td>
<td>CC</td>
</tr>
<tr>
<td>RW3</td>
<td>${P_1 \land \neg P_5, \neg P_3 \land P_2}$</td>
<td>FPC</td>
</tr>
<tr>
<td>RW4</td>
<td>${P_1 \land \neg P_5, \neg P_3 \land P_2, P_3 \land P_2}$</td>
<td>MCC</td>
</tr>
</tbody>
</table>

Table 1. Disjunction Rewritings and Coverage Criteria

Finally, the last rewriting evaluates each possibility to satisfy the disjunction. This allows to satisfy the Multiple Condition Coverage (MCC).

Here again, we can establish the following hierarchy between the rewriting and the condition coverage criteria.

$$RW1 \subseteq RW2 \subseteq RW3 \subseteq RW4$$

Notice that measuring the coverage of a precondition can be reduced to measuring the satisfaction of the precondition, whereas usually the unsatisfaction of the precondition is also measured. Since we independently consider the negation of the precondition, by construction of the graph, this step is implicitly performed.

For practical reasons, all these rewritings are only applied on the positive preconditions of the method specification blocks. Indeed, the application of these rewritings on negations of the preconditions would lead to a combinatorial explosion of the number of cases. Nevertheless, it is possible to apply RW1 or RW2, which may be an indicator of whether the test suite tries to perform unauthorized actions, and the contexts these actions are tried to be activated.

5 Performing measurements

First, we introduce the principles used to perform the coverage measurement. Second, we present a tool implementing these principles.

5.1 Principles

The principle of checking the coverage of a JML specification is similar to the run-time checking of the assertions as performed in the RAC. It is presented as a preprocessing which enriches the original Java code with the verification of the JML predicates. In addition, we need to setup a Coverage Report Manager (CRM) dedicated to the measures must be performed.

The CRM keeps track of the graphs representing each method specification. Each time a predicate is checked within the source code, the report manager is informed of the edge that has been covered and the node that has been reached. In brief, the principle is illustrated on a generic method specification in Fig. 5.

It is important to notice that, as for the JML Runtime Assertion Checker, the verifications added to the Java code do not change the functional behavior of the methods and the functional behavior of the program in general.
/\* @ behavior
  @ requires \( P_i \);
  @ assignable \( A_i \);
  @ ensures \( Q_i \);
  @ signals \( (E_{i1} \text{ or} \ 1) \ S_{i1} \);
  @ ... 
  @ signals \( (E_{iM} \text{ or} \ 1) \ S_{iM} \);
  @ also 
  @ ... 
  @ also 
  @ requires \( P_N \);
  @ assignable \( A_i \);
  @ ensures \( Q_N \);
  @ signals \( (E_{N1} \text{ or} \ 1) \ S_{N1} \);
  @ ... 
  @ signals \( (E_{NP} \text{ or} \ 1) \ S_{NP} \); 
@/

Type meth(T; p1, ...) throws E11, ... {
  try {
    // Check and report precondition
    // edges predicates coverage
    body;
  }
  catch(java.lang.Error e) {
    if (e instanceof E11) {
      // Check and report edges
      // predicates coverage for E1
    }
    ... 
    if (e instanceof ENP) {
      // Check and report edges
      // predicates coverage for E_N
    }
    throw e;
  }
  // Check and report edges predicates coverage for normal postcondition
} 

Fig. 5. Instrumented Java source code

Dedicated internal methods, within the CRM are in charge of computing the edges/nodes coverage achieved at the end of the test suite execution. It is possible to display a report or to consult them using a customized API.

5.2 The jmlCoverage Tool

The jmlCoverage tool implements the principles described before, as illustrated in Fig. 6. It acts as a precompiler that produces the Coverage Report Manager (as a Java source file) and the monitor itself, as an AspectJ file or an instrumented
Java source file, that is in charge of monitoring the execution of the observed
methods.

When the main program execution is over, the Coverage Report Manager
displays a table that informs of the coverage of the nodes/edges/paths of the JML
method specification graph, for each condition rewriting that can be applied.

jmlCoverage has been developed, by choice, independently from the JML Run-
time Assertion Checker. The tool supports the same functionalities as the RAC
and, as a consequence, it requires the JML expressions to be executable (i.e.,
by iterating \forall or \exists over a finite range of integers). Basically, all
constructs accepted by the JML RAC can be accepted by the tool. Its use is inde-
pendent from the RAC and, whereas it is not its first intent, jmlCoverage is also
able to detect postconditions that are not established by the implementation.

The next section reports on the use of the jmlCoverage tool within a realistic
case study.

5.3 Experiments

Target program. We have experimented our approach on a case study, adapted
from an industrial example, named Demoney [19]. Demoney is an applet designed
by Trusted Logics, implementing an electronic purse. For the experimental pur-
pose, we have developed a simplified version of the implementation, which had
been previously annotated with JML to describe its functional behavior. The
classes of the application represent about 500 lines of JML spread in 4 classes.

Selected testing tools. Then we have selected two (semi-)automated test gen-
eration tools, for which we wanted to evaluate the test suite generation capabil-
ities. We have selected a random testing tool, JARTEGE [23], and a combinatorial
testing tool, TOBIAS [18]. JARTEGE produces a given number of sequences, each
sequence being of a given length, and composed of randomly selected method
invocations using random inputs. On the other hand, TOBIAS is able to produce
large combinatorial test suites from a test schema defined as a regular expres-
sion. Both tools rely on the JML method specifications to filter test cases that
do not fulfill the method preconditions.

Study. First, we ran JARTEGE on the Demoney class. Since JARTEGE is a ran-
dom testing tool, we were interested in evaluating the efficiency of such a tool. Its
use shows the practicability of our approach, as well as an interesting feedback
on the produced test suites. Indeed, the possibility to connect JARTEGE with
jmlCoverage to help has appeared to be an interesting option. In this context,
jmlCoverage can be used to limit the number of generated test cases, by gen-
nerating tests until a user-defined specification coverage rate is reached. Second, we
designed 5 testing schemas that TOBIAS unfolded in 162 test cases. The resulting
abstract tests were concretized to a Java test program. We have been able to
establish the overall coverage of our testing schema. Here again, our tool can
be used to master the combinatorial explosion induced by the use of schemas.
Table 2. Results of execution times on the case study (in ms.)

<table>
<thead>
<tr>
<th>TS size × number of TS</th>
<th>javac</th>
<th>RAC</th>
<th>jmlCoverage</th>
<th>RAC + jmlCoverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>100 × 10</td>
<td>0.738</td>
<td>1.304</td>
<td>0.768</td>
<td>1.244</td>
</tr>
<tr>
<td>100 × 20</td>
<td>1.195</td>
<td>1.116</td>
<td>1.140</td>
<td>1.676</td>
</tr>
<tr>
<td>100 × 50</td>
<td>2.671</td>
<td>1.165</td>
<td>1.711</td>
<td>2.010</td>
</tr>
<tr>
<td>100 × 100</td>
<td>4.987</td>
<td>1.565</td>
<td>2.109</td>
<td>2.464</td>
</tr>
</tbody>
</table>

For both of these tools, the specification coverage measure is a good help for a validation engineer to know whether the test suites are pertinent or not.

One interesting point is the comparison of the execution times of the test suites w.r.t. the additional annotations. Table 2 displays the execution times (in ms) of several test suites w.r.t. on programs (i) without any runtime verification, (ii) with the JML assertions checking, (iii) with the JML assertions coverage measure, and (iv) with (ii) and (iii). Notice that the test suites were automatically generated using JARTEGE. Notice also that execution times with (i) may be longer than in other cases, since the runtime checking may reveal inconclusive tests (i.e., which do not respect one method’s precondition), and interrupts the execution of the corresponding test suite.

The results show that the cost of executing jmlCoverage is very little, regardless to executing the RAC. This is due to the fact that the RAC performs lot more checkings than jmlCoverage, since it systematically checks invariants and history constraints. But, the additional cost of using jmlCoverage on top of the RAC is minimal, even for larger test suites.

6 Related Work

The JML Runtime Assertion Checker [5] is already able to report a partial coverage of a JML method specification, indicating if a precondition has been covered once, more than once, or never. Nevertheless, it does not present the same granularity as our approach and can not be considered as a relevant coverage measure tool. VDMTools [11] also adopt a DBC approach. They provide coverage tools which consider pre- and postconditions as ordinary statements and measure how much of the specification has been exercised. In other words, it provides an extended notion of statement coverage which is most of the time weaker than our measures. Works also been led on the coverage measure of UML specifications [1], especially based on the structural coverage of statecharts diagrams. Simulink Stateflow [12] is also able to perform model coverage measurement on statecharts diagrams. A complementary view of test suite measurement is the code coverage measurement, that can be achieved with tools such as JCover [14], JCoverage [15], clover [7] or EMMA [10].

The approach we have proposed is inspired of both classical control-flow graph coverage criteria [21], and classical condition coverage criteria [22]. The novelty is the application of these criteria to a predicate-based graph extracted from a JML method specification. Moreover, the interest of using a specification coverage tool instead of a code coverage is that the specification makes it possible to express properties independently from a specific implementation, and thus, allows more specific measurements, based on a black-box view of the program.
7 Conclusion and Future Works

This paper has presented an approach for measuring the coverage of JML method specifications by a Java test suite. A run-time assertion checking mechanism is employed to ensure the coverage of the graph extracted from the method specifications. The originality of this work is the application of the criteria to JML. We believe that this work can help increasing the confidence of a validation engineer in his/her test suite, even if it does not replace a code coverage analysis. From a technical point of view, the use of aspects for runtime checking of the assertions, frees us from requesting the Java source code. We only need the JML specification. As a consequence, this approach is suited to model-based testing.

The work presented in this paper can be used as a basis for reducing test suites w.r.t. a defined coverage criterion so that the reduced test suite provides the same coverage as the complete one [13]. Moreover, the Java interface could be interesting for connecting JARTEGE, also written in Java.

One interesting point is to extend the coverage of the JML specifications to take other clauses into account, such as the class invariant or the history constraints. In addition, we would like to base the development of jmlCoverage on the RAC’s architecture. This would increase the evolutions of the tool w.r.t. the evolutions of JML, and it would make it possible to reuse the assertion generation mechanisms of the RAC. Finally, the use of an annotation modeling language such as JML, leads us to consider the extension of this work to Spec# specifications, which would probably not present any difficulties.

References


28
Abstract. Several model-checker based methods to automated test-case generation have been proposed recently. The performance and applicability largely depends on the complexity of the model in use. For complex models, the costs of creating a full test-suite can be significant. If the model is changed, then in general the test-suite is completely regenerated. However, only a subset of a test-suite might be invalidated by a model change. Creating a full test-suite in such a case would therefore waste time by unnecessarily recreating valid test-cases. This paper investigates methods to reduce the effort of recreating test-suites after a model is changed. This is also related to regression testing, where the number of test-cases necessary after a change should be minimized. This paper presents and evaluates methods to identify obsolete test-cases, and to extend any given test-case generation approach based on model-checkers in order to create test-cases for test-suite update or regression testing.

1 Introduction

The need for efficient methods to ensure software correctness has resulted in many different approaches to testing. Recently, model-checkers have been considered for test-case generation use in several works. In general, the counter example mechanism of model-checkers is exploited in order to create traces that can be used as test-cases.

If the model used for test-case generation is changed, this has several effects. Test-cases created with a model-checker are finite execution paths of the model, therefore a test-suite created previously to the model change is likely to be invalid. As test-case generation with model-checkers is fully automated the obvious solution would be to create a new test-suite with the changed model. This is a feasible approach as long as the model complexity is small. If the model complexity is significant, the use of a model-checker can lead to high computational
costs for the test-case generation process. However, not all of the test-cases might be invalidated by the model change. Many test-cases can be valid for both the original and the changed model. In that case, the effort spent to recreate these test-cases would be wasted. There are potential savings when identifying invalid test-cases and only creating as many new test-cases as necessary. If a model is changed in a regression testing scenario, where a test-suite derived from the model before the change fails to detect any faults, running a complete test-suite might not be necessary. Here it would be sufficient to run those tests created with regard to the model change.

In this paper, we present different approaches to handle model changes. Which approach is preferable depends upon the overall objectives in a concrete scenario - should the costs be minimized with regard to test-case generation or test-case execution, or is it more important to ensure that the changes are correctly implemented? The contributions of this paper are as follows:

- We present methods to decide whether a test-case is made obsolete by a model change or if it remains valid. The availability of such a method allows the reuse of test-cases of an older test-suite and is a necessary prerequisite to reduce the costs of the test-case generation process for the new model.
- We present different methods to create new test-cases after a model change. These test-cases can be used as regression tests if the number of test-cases executed after a model change should be minimized. They are also used in order to update test-suites created with older versions of the model.
- An empirical evaluation tries to answer two research questions: 1) What is the impact of test-suite update on the overall quality compared to newly created test-suites? 2) Is there a performance gain compared to completely re-creating a test-suite after a model change?

This paper is organized as follows: Section 2 identifies the relevant types of changes to models and relates them to a concrete model-checker input language, and then presents our solutions to the tasks of identifying invalid test-cases and creating new ones. Section 3 describes the experiment setup and measurement methods applied to evaluate these approaches, and presents the results of these experiments. Finally, Section 4 discusses the results and concludes the paper.

2 Handling Model Changes

As this paper considers test-cases created with model-checkers, this section recalls the basic principles of such approaches. After a short theoretical view of model changes we use the input language of a concrete model-checker to present methods to handle model changes.

2.1 Preliminaries

A model-checker is a tool used for formal verification. It takes as input an automaton model and a temporal logic property and then effectively explores the
entire state space of the model in order to determine whether model and property are consistent. If inconsistency is detected, the model-checker returns an example execution trace (counter example) that illustrates how a violating state can be reached. The idea of model-checker based testing is to use these counter examples as test-cases. Several different approaches of how to force the model-checker to create traces have been suggested in recent years. Model-checkers use Kripke structures as model formalism:

**Definition 1.** Kripke Structure: A Kripke structure $M = (S, s_0, T, L)$, where $S$ is the set of states, $s_0 \in S$ is the initial state, $T \subseteq S \times S$ is the transition relation, and $L : S \rightarrow 2^{AP}$ is the labeling function that maps each state to a set of atomic propositions that hold in this state. $AP$ is the countable set of atomic propositions.

Properties are specified with temporal logics. In this paper we use Linear Temporal Logic (LTL) [1]. An LTL formula consists of atomic propositions, Boolean operators and temporal operators. $X$ refers to the next state. For example, $X a$ expresses that $a$ has to be true in the next state. $U$ is the until operator, where $a U b$ means that $a$ has to hold from the current state up to a state where $b$ is true. Other operators can be expressed with the operators $U$ and $X$. E.g., $G x$ is a shorthand for $-(true U \neg x)$ and requires $x$ to be true at all times. The syntax of LTL is given as follows, with $a \in AP$:

$$
\phi ::= a | \neg \phi | \phi_1 \land \phi_2 | X\phi | \phi_1 U \phi_2 | G\phi.
$$

If the model-checker determines that a model $M$ violates a property $\phi$ then it returns a trace that illustrates the property violation. The trace is a finite prefix of an execution sequence of the model (path):

**Definition 2.** Path: A path $p := \langle s_0, ..., s_n \rangle$ of Kripke structure $M$ is a finite or infinite sequence such that $\forall 0 \leq i < n : (s_i, s_{i+1}) \in T$ for $M$.

A test-case $t$ is a finite prefix of a path $p$. We consider such test-cases where the expected correct output is included. This kind of test-cases is referred to as passing or positive test-cases. The result of the test-case generation is a test-suite. As test-cases created by model-checkers are linear sequences, they cannot account for non-deterministic behavior of the system under test. We therefore restrict the presented results to deterministic models. The main application area of model-checker based testing are reactive systems, where inputs are processed in a cyclic way and output values are set accordingly.

**Definition 3.** A test-case $t$ for model $M = (S, s_0, T, L)$ is invalid for the altered model $M' = (S', s_0', T', L')$, if any of the following conditions is true:

$$
\exists i : < ..., s_i, s_{i+1}, ... > = t \land (s_i, s_{i+1}) \notin T' \quad (1)
$$

$$
\exists i : < ..., s_i, ... > = t \land L(s_i) \neq L'(s_i) \quad (2)
$$

$$
\exists i : < ..., s_i, ... > = t \land (s_i \notin S') \quad (3)
$$

In practice, the Kripke structure is described with the input language of the model-checker in use. Such input languages usually describe the transition relation by defining conditions on $AP$, and setting the values of variables according
to these conditions. A transition condition \( C \) describes a set of states \( S_i \) where \( C \) is fulfilled. In all successor states of these states the variable \( v \) has to have the next value \( n \): \( \forall s \in S_i : L(s) \models C \land \forall s' : (s, s') \in T \rightarrow "v = n" \in L(s'). \) In this paper, we use the syntax of the model-checker NuSMV [2]. Listing 1 shows how a transition relation looks like in NuSMV models. A change in the Kripke structure is represented by a syntactical change in the model source. Such changes can be automatically detected, e.g., by a comparison of the syntax trees. We are only interested in changes that do not invalidate a complete test-suite. Traces created by a model-checker consist only of states \( s \) such that for each variable \( v \) defined in the model source there exists the proposition "\( v = n \)" \( \in L(s) \), where \( n \) is the current value of variable \( v \) in state \( s \). For example, addition or removal of a variable in the model source would result in a change of \( L \) for every state in \( S \), and would therefore invalidate any test-suite created before the change. Consequently, the interesting types of changes are those applied to the transition conditions or the values for the next states of variables in the model description.

\[
\text{ASSIGN} \quad \text{next}(\text{var}) := \text{case} \quad \text{init}(x) := 1; \\
\quad \text{condition}_1: \text{next_value}_1; \quad \text{next}(x) := \text{case} \quad \text{next}(x) := \text{case} \\
\quad \text{condition}_2: \text{next_value}_2; \quad \text{State} = 0: 0; \quad \text{State} = 1: 1; \\
\text{esac}; \quad 1: x; \quad \text{esac}; \\
\text{Listing 1. ASSIGN section of an SMV file.} \\
\text{Listing 2. Transition relation of } \quad t := (\langle x = 1 \rangle, \langle x = 0 \rangle, \langle x = 1 \rangle). \\
\]

2.2 Identifying Obsolete Test-Cases

In order to use a model-checker to decide if a test-case is valid for a given model, the test-case is converted to a verifiable model. The transition relations of all variables are given such that they depend on a special state counting variable, as suggested by Ammann and Black [3]. An example transition relation is modeled in Listing 2, where \( \text{State} \) denotes the state counting variable. There are two methods to decide whether a test-case is still valid after a model change. One is based on an execution of the test-case on the model, and the other verifies change related properties on the test-case model.

\textbf{Symbolic Test-case Execution:} A model-checker is not strictly necessary for symbolic execution. However, in a scenario of model-checker based test-case generation the possibility to use a model-checker is convenient, as it avoids the need for an executable model. Symbolic execution of a test-case with a model-checker is done by adding the actual model as a sub-model instantiated in the test-case model. As input variables of the sub-model the values of the test-case are used. Finally, by verifying a property that claims that the output values
of the test-case and the sub-model equal for the length of the test-case, the model-checker determines whether this is indeed the case:

```plaintext
MODULE changed_model(input variables)
Transition relation of changed model

MODULE main
test-case model
VAR
SubModel: changed_model(input vars);
SPEC G(State < max_state -> output = SubModel.output)
```

Now the problem of checking the validity of a test-suite with regard to a changed model reduces to model-checking each of the test-cases combined with the new model. Each test-case that results in a counter example is obsolete. The test-cases that do not result in a counter example are still valid, and thus are not affected by the model change. A drawback of this approach is that the actual model is involved in model-checking. If the model is complex, this can have a severe impact on the performance.

**Change Properties:** In many cases, test-case models can simply be checked against certain properties in order to determine whether a change has an influence on a test-case’s validity. This avoids the inclusion of the new model in the model-checking process. If a transition condition or target is changed, then the changed transition can be represented as a temporal logic property, such that any test-case model that is valid on the new model has to fulfill the property: $G(changed\_condition \rightarrow X variable = changed\_value)$. Such change properties can be created automatically from the model-checker model source file. The concrete method depends on the syntax used by the model-checker.

If a variable transition is removed, it can only be determined whether a test-case takes the old transition using a negated property: $G(old\_condition \rightarrow X!(variable = old\_value))$. Any test-case that takes the old transition results in a counter example.

Theoretically, the latter case can report false positives, if the removed transition is subsumed or replaced by another transition that behaves identically. This is conceivable as a result of manual model editing. Such false positives can be avoided by checking the new model against this change property. Only if this results in a counter example the removal has an effect and really needs to be checked on test-cases. Although verification using the full model is necessary, it only has to be done once in contrast to the symbolic execution method.

Test-cases that are invalidated by a model change can be useful when testing an implementation with regard to the model change. Obsolete positive test-cases can be used as negative regression test-cases. A negative test-case is such a test-case that may not be passed by a correct implementation. Therefore, an implementation that passes a negative regression test-case adheres to the behavior described by the old model.
2.3 Creating New Test-Cases

Once the obsolete test-cases after a model change have been identified and discarded, the test-cases that remain are those that exercise only unchanged behavior. This means that any new behavior added through the change is not tested. Therefore, new test-cases have to be created.

Adapting Obsolete Test-Cases: Analysis of the old test-suite identifies test-cases that contain behavior that has been changed. New test-cases can be created by executing these test-cases on the changed model, recording the new behavior. This is done with a model-checker by combining test-case model and changed model together as described in Section 2.2. The test-case model contains a state counter $\text{State}$, and a maximum value $\text{MAX}$. The model-checker is queried with the property $\text{G}\,(\text{State} \neq \text{MAX})$. This achieves a trace where the value of $\text{State}$ is increased up to $\text{MAX}$. The adapted test-case simply consists of the value assignments of the changed model in that trace.

Alternatively, when checking test-cases using the symbolic execution method, the counter examples in this process can directly be used as test-cases. In contrast to the method just described resulting test-cases can potentially be shorter, depending on the change. This can theoretically have a negative influence on the overall coverage of the new test-suite.

The drawback of this approach is that the changed model might contain new behavior which cannot be covered if there are no related obsolete test-cases. In the evaluation we refer to this method as Adaptation.

Selectively Creating Test-Cases: Xu et al. [4] presented an approach to regression testing with model-checkers, where a special comparator creates trap properties from two versions of a model. In general, trap property based approaches to test-case generation express the items that make up a coverage criterion as properties that claim the items cannot be reached [5]. For example, a trap property might claim that a certain state or transition is never reached. When checking a model against a trap property the model-checker returns a counter example that can be used as a test-case. We generalize the approach of Xu et al. in order to be applicable to a broader range of test-case generation techniques. The majority of approaches works by either creating a set of trap properties or by creating mutants of the model.

For all approaches using trap properties we simply calculate the difference of the sets of trap properties, as an alternative to requiring a special comparator for a specific specification language and coverage criterion. The original model results in a set of properties $P$, and the changed model results in $P'$. New test-cases are created by model-checking the changed model against all properties in $P' - P$. The calculation of the set difference does not require any adaptation of given test-case generation frameworks. In addition, it also applies to methods that are not based on coverage criteria, e.g., the approach proposed by Black [6]. Here, properties are generated by reflecting the transition relation of the SMV source file as properties similar to change properties presented in Section 2.2. The resulting properties are mutated, and the mutants serve as trap properties.
It is conceivable that this approach might not guarantee achievement of a certain coverage criterion, because for some coverable items the related test-cases are invalidated, even though the item itself is not affected by the change. If maximum coverage of some criterion is required, then an alternative solution would be to model-check the test-case models against the set of trap properties for the new model instead of selecting the set difference. For reasons of simplicity, we consider the straightforward approach of using set differences in this paper.

In the evaluation we refer to this method as \textit{Update}.

The second category of test-case generation approaches uses mutants of the model to create test-cases (e.g., \cite{7-9}). For example, state machine duplication \cite{8} combines original and mutant model so that they share the same input variables. The model-checker is then queried whether there exists a state where the output values of model and mutant differ. Here, the solution is to use only those mutants that are related to the model change. For this, the locations of the changes are determined (e.g., in the syntax tree created by parsing the models) and then the full set of mutants for the changed model is filtered such that only mutants of changed statements in the NuSMV source remain. Test-case generation is then performed only using the remaining mutants.

\textbf{Testing with Focus on Model Changes} As a third method to create change related test-cases we propose a generic extension applicable to any test-case generation method. It rewrites both the model (or mutants thereof) and properties involved in the test-case generation just before the model-checker is called. This rewriting is fully automated. The model is extended by a new Boolean variable \texttt{changed}. If there is more than one change, then there is one variable for each change: \texttt{changer}. These variables are initialized with the value false. A change variable is set to true when a state is reached where a changed transition is taken. Once a change variable is true, it keeps that value. The transition relation of the change variable consists of the transition condition of the changed variable:

\begin{verbatim}
MODULE main
VAR
    changed: boolean;
...
ASSIGN
    init(changed) := FALSE;
    next(changed) := case
        changed_con: TRUE;
        1: changed; -- default branch
    esac;
    next(changed_var) := case
        changed_con: changed_value;
    esac;
...
\end{verbatim}

The properties involved in the test-case generation approach are rewritten in order to create test-cases with focus on the model change. As an example we use LTL, although the transformation can also be applied to computation tree logic (CTL) \cite{10}.
Definition 4. Change Transformation: The change transformation $\phi' = \alpha(\phi, c)$ for an LTL property $\phi$ with respect to the change identified with the Boolean variable $c$, with $a \in AP$ being a propositional formula, is recursively defined as:

\[
\begin{align*}
\alpha(a, c) &= a \quad (4) \\
\alpha(\neg \phi, c) &= \neg \alpha(\phi, c) \quad (5) \\
\alpha(\phi_1 \land \phi_2, c) &= \alpha(\phi_1, c) \land \alpha(\phi_2, c) \quad (6) \\
\alpha(X\phi, c) &= X(c \rightarrow \alpha(\phi, c)) \quad (7) \\
\alpha(\phi_1 U \phi_2, c) &= \alpha(\phi_1, c) U (c \rightarrow \alpha(\phi_2, c)) \quad (8)
\end{align*}
\]

Basically, all temporal operators are rewritten to include an implication on the change variable. This achieves that only such counter examples are created that include the changed transition. For multiple changes there has to be one modified version of each property for each change in order to make sure that all changes are equally tested. In the evaluation we refer to this method as Focus.

3 Empirical Results

The previous section presented different possibilities for different aims to cope with model changes in a scenario of model-checker based test-case generation. This section tries to evaluate the feasibility of these ideas. First, the experiments conducted are described, and then the results are presented and discussed.

3.1 Experiment Setup

The methods described in this paper have been implemented using the programming language Python and the model-checker NuSMV [2]. All experiments have been run on a PC with Intel Core Duo T2400 processor and 1GB RAM, running GNU/Linux. We automatically identify changes between two versions of a model by an analysis of the abstract syntax trees created from parsing the models. We use a simple example model of a cruise control application based on a version by Kirby et al. [11]. In order to evaluate the presented methods, the mutation score and creation time of new and updated test-suites were tracked over several changes. There is a threat to the validity of the experiments by choosing changes that are not representative of real changes. Therefore the experiments were run several times with different changes and the resulting values are averaged.

In the first step, mutants were created from the supposedly correct model. The following mutation operators were used (see [12] for details): StuckAt (replace atomic propositions with true/false), ExpressionNegation (negate atomic propositions), Remove (remove atomic propositions), LogicalOperatorReplacement, RelationalOperatorReplacement, ArithmeticOperatorReplacement. The resulting mutants were analyzed in order to eliminate equivalent mutants. This is done with a variant of the state machine duplication approach [8], where the model-checker is queried whether there exists a state where the output values of
a model and its mutant differ. An equivalent mutant is detected if no counterexample is returned. The set of mutants was further reduced by checking each mutant against a set of basic properties that require some elementary behavior, e.g., reachability of some important states.

Out of the resulting set of inequivalent mutants one mutant is chosen randomly, and used as new model. With this mutant, the procedure is repeated until a sequence of 20 visible model changes is achieved. The experiment was run on 20 such sequences and the results were averaged.

For each of the sequences of model versions the following is performed: Beginning with the model containing all 20 changes, test-suites are created using the methods transition coverage criterion (one test-case for each transition condition of the NuSMV model), mutation of reflected transition relation [6] and state machine duplication [8]. These three methods were chosen as they should be representative for most types of conceivable approaches. Then, the next version of the model is chosen, and the test-suites of the previous model are analyzed for obsolete test-cases, and new and updated test-suites are created. Then the mutation scores of all of these test-suites are calculated. The mutation score is the ratio of identified mutants to mutants in total. It is calculated by symbolically executing the test-case models against the mutant models. This procedure is repeated for each of the model versions up to the original model.

3.2 Results

![Fig. 1. Transition coverage test-case generation.](image)

Figures 1-3(a) show the mutation scores of the different methods along the course of the different model version. There is a degradation of the mutation score for the adaptation and update methods. The degradation increases with each model change, therefore it could be advisable to create new test-suites
after a certain number of changes when using such a method. In contrast, the change focus method achieves a mutation score that is sometimes even higher than that of a completely new test-suite. This is because the test-suites created with the focus method are bigger than new test-suites for the transition coverage criterion. Adaptation generally achieves the lowest mutation scores. However, the mutation score is only slightly smaller than for the update method, so a significant performance gain could justify this degradation.

Figures 1-3(b) show the computation times for creating and for updating test-suites. All methods are faster than a complete test-case generation process. Test-suite adaptation performs significantly faster than all other methods in most cases. The performance of the adaptation is determined by the model complexity and number of invalid test-cases, and is therefore similar for all test-suites in the experiment. In contrast, the performance of the update and focus...
methods depend on the test-case generation approach they are based upon. For simple methods like transition coverage focus is very efficient, while there is less performance gain as test-case generation complexity increases. For the most complex approach used in the experiment (based on state machine duplication) the performance gain in comparison to creating a new test-suite is minimal.

Finally, Table 1 compares the numbers of new test-cases created in average after each model change. This chart reveals why the change focus method achieves such high mutation scores: the number of test-cases generated is significantly higher than for any other approach. Interestingly it is even higher than the number of test-cases generated for new test-suites with the transition and reflection approaches, although the test-case generation is still faster in average.

<table>
<thead>
<tr>
<th>Test-Suite Type</th>
<th>Full</th>
<th>Adaptation</th>
<th>Update</th>
<th>Focus</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transition</td>
<td>5.75</td>
<td>1</td>
<td>1.4</td>
<td>6.6</td>
</tr>
<tr>
<td>Reflection</td>
<td>19.35</td>
<td>2.5</td>
<td>7.05</td>
<td>24.4</td>
</tr>
<tr>
<td>SM Duplication</td>
<td>33.35</td>
<td>2.85</td>
<td>6.45</td>
<td>29.4</td>
</tr>
</tbody>
</table>

Table 1. Average number of unique new test-cases.

## 4 Conclusion

In this paper, we have shown how to decide whether a test-case is still valid after the model it was created from is changed. That way, it is possible to reuse some of the test-cases after a model change and reduce the test-suite generation effort. Different methods to create test-cases specific to the model change were presented. We used the model-checker NuSMV for our experiments and as an example model syntax. However, there is no reason why the approach should not be applicable to other model-checkers. Experiments have shown that the presented methods can be used to update test-suites after a model change, although there is a trade-off between performance improvement and quality loss.

The main problem of model-checker based approaches in general is the performance. If the model is too complex, then test-case generation will take very long or might even be impossible. Therefore, it is important to find ways of optimizing the approach. The potential savings when recreating test-suites after a model change are significant. Even for the small model used in our evaluation a large performance gain is observable when only selectively creating test-cases for the model changes. Although a model is usually more abstract than the program it represents, the model size can still be significant. For instance, automatic conversion (e.g., Matlab Stateflow to SMV) can result in complex models.

The methods presented to create new test-cases with minimal computational effort achieve good results. We cannot conclude that one method is superior, because the preferable method depends on the concrete scenario. If the main goal is to minimize the costs of retesting, then adaptation of old test-cases is effective as long as there are not too many and significant changes. If it is more important to maximize likelihood of detecting faults with relation to the change, then the
presented method to create test-cases focusing on a change is preferable. For example, in safety related scenarios a decrease of the test-suite quality is unacceptable. Finally, the update method that creates test-cases only for changed parts seems like a good compromise; it reduces the costs while the quality decrease is not too drastic. Test-cases created with any of the presented methods can be used as regression test-suites, following the ideas of Xu et al. [4].

There are some approaches that explicitly use specification properties for test-case generation [7–9]. This paper did not explicitly cover the aspects of test-suite update with regard to specification properties. However, the idea of test-suite focus directly applies to such approaches, as well as the presented test-suite update techniques. The cruise control example is only a small model, and the changes involved in our experiments were generated automatically. This is sufficient to show the feasibility of the approach. However, actual performance measurements on more complex models and realistic changes would be desirable.

References


A Global Algorithm for Model-Based Test Suite Generation

Anders Hessel1 and Paul Pettersson1,2

1 Department of Information Technology, Uppsala University, P.O. Box 337, SE-751 05 Uppsala, Sweden. E-mail: {hessel,paupet}@it.uu.se.
2 Department of Computer Science and Electronics, Mälardalen University, P.O. Box 883, SE-721 23, Västeras, Sweden. E-mail: Paul.Pettersson@mdh.se.

Abstract. Model-based testing has been proposed as a technique to automatically verify that a system conforms to its specification. A popular approach is to use a model-checker to produce a set of test cases by formulating the test generation problem as a reachability problem. To guide the selection of test cases, a coverage criterion is often used. A coverage criterion can be seen as a set of items to be covered, called coverage items. We propose an on-the-fly algorithm that generates a test suite that covers all feasible coverage items. The algorithm returns a set of traces that includes a path fulfilling each item, without including redundant paths. The reachability algorithm explores a state only if it might increase the total coverage. The decision is global in the sense that it does not only regard each individual local search branch in isolation, but the total coverage in all branches together. For simpler coverage criteria as location of edge coverage, this implies that each model state is never explored twice.

The algorithm presented in this paper has been implemented in the test generation tool UPPAL COVER. We present encouraging results from applying the tool to a set of experiments and in an industrial sized case study.

1 Introduction

The bulk of verification efforts in software industry today is performed using various testing techniques. In conformance testing, the behavior of an implemented system, or system part, is checked to agree with its specification. This is typically done in a controlled environment where the system is executed and stimulated with input according to a test specification, and the responses of the system are checked to conform to its specification. To reduce the costs of this process, the execution of software testing is often automated, whereas the production of test suites are mostly done by hand. Techniques to automatically generate test suites, or to combine generation and execution, are emerging and getting more mature [31,9,28,19].

In this paper, we study techniques for model-based conformance testing in a setting where the test suite is automatically generated from a model before the actual testing takes place — sometimes referred to as offline testing in contrast to online testing [23]. In order to guide the generation of tests and to describe how thorough the tests should be, we select tests following a particular coverage criterion, such as coverage of control states or edges in a model. Many coverage criteria have been suggested in the
literature [27,6,12] ranging from simple structural criteria to complex data-flow criteria characterized as path properties. Many algorithms for generating test suites following a given coverage criterion have also been proposed [29,22,18,13], including algorithms producing test suites optimal in the number of test cases, in the total length of the test suite, or in the total time required to execute the test suite.

In this paper, we study test suite generation algorithms inspired by reachability analysis techniques used in model-checkers such as SPIN [16] and UPPAAL [24] — an approach shared with, e.g., [19]. Such algorithms essentially perform reachability analysis to generate and explore the state space of a model in order to find a set of paths that follows a given coverage criterion, which can be interpreted as a test suite. To generate a path, a coverage criterion can be regarded as a set of independent coverage tasks [11] or coverage items [4] to be covered. Reachability analysis is applied to generate a set of paths for all reachable coverage items. We review this technique and suggest a number of modifications to improve the efficiency of the analysis.

The main contribution of this paper is a novel on-the-fly algorithm for generating test suites by reachability analysis. It can be seen as a trade-off between performance of the algorithm, in terms of time and space requirements, and generating a test suite with reasonable characteristics. The result is an algorithm that in each step uses global information about the state space generated so far to guide the further analysis and to speed up termination. The generated test suite is reasonable in the sense that each coverage item is reached by a path from the initial state to the first found state in which it is satisfied.

During the state-space exploration, the algorithm stores a set of paths to the coverage items satisfied so far. This information is used to prune search branches that will not be able to contribute to the total coverage — a technique that improves the performance of the algorithm. In experiments we justify this statement by presenting how the algorithm, implemented in the UPPAAL CO\textsc{\small{ER}} tool \footnote{See the web page \url{http://user.it.uu.se/~hessel/CoVer/} for more information about the UPPAAL CO\textsc{\small{ER}} tool.}, performs on a set of examples from the literature.

The rest of the paper is organized as follows: in Section 2 we describe the model used in this paper, and review techniques for test case generation based on reachability analysis. In Section 3 we describe a reachability analysis algorithm for test case generation. In Section 4 we present a novel algorithm for test case generation that uses global information about the generated state-space to determine termination and pruning. In Section 5 we describe the results of experiments comparing the different techniques. The paper ends with conclusions in Section 6.

Related Work: Our work is mostly related to test case generation approaches inspired by model-checking techniques, including [5,13,23,19,17,28]. In [28], Nielsen and Skou generate test cases that cover symbolic states of Event Recording Automata. Like our work, the proposed state-space exploration algorithm is inspired by model-checking, however the work is focused on timed system and uses a fixed coverage criterion.
In [19], Hong et al show how several flow-based coverage criteria can be expressed in temporal logic and how the test case generation problem can be solved by model-checking. Hong and Ural [17] continue this work and study how coverage items can subsume each other, and propose a solution to the problem. These works use an existing CTL model-checker to solve the test case generation problem, whereas we propose a specialized algorithm for test case generation.

Our work is also related to directed model-checking techniques, where state-space exploration is guided by the property to be checked. In [8], the authors use a bi-state hashing based iterated search refinement method to guide a model-checker to generate test cases. This method can be seen as a meta algorithm using an existing model-checker iteratively. Thus the actual model-checking algorithms is not refined for test case generation.

2 Preliminaries

We will present ideas and algorithms for test case generation applicable to several automata based models, such as finite state machines, extended finite state machines (EFSM) as, e.g., SDL [20], or timed automata [1]. Throughout this paper, we shall present our results using the model of communicating EFSMs.

2.1 The Model

An EFSM $F$ over actions $Act$ is a tuple $\langle L, l_0, V, E \rangle$, where $L$ is a set of locations, $l_0 \in L$ the initial location, $V$ is a finite set of variables with finite value domains, and $E$ is a set of edges. An edge is of the form $\langle l, g, \alpha, u, l' \rangle \in E$, where $l \in L$ is the source location and $l' \in L$ the destination location, $g$ is a guard (a predicate) over $V$, $\alpha \in Act$ an action, and $u$ is an update in the form of an assignment of variables in $V$ to expressions over $V$.

A state of an EFSM is a tuple $\langle l, \sigma \rangle$ where $l \in L$ and $\sigma$ is a mapping from $V$ to values. The initial state is $\langle l_0, \sigma_0 \rangle$ where $\sigma_0$ is the initial mapping. A transition is of the form $\langle l, \sigma \rangle \xrightarrow{\alpha, u} \langle l', \sigma' \rangle$ and is possible if there is an edge $\langle l, g, \alpha, u, l' \rangle \in E$ where the $g$ is satisfied for the valuation $\sigma$, the result of updating $\sigma$ according to $u$ is $\sigma'$, and $\alpha$ is an action.

A network of communicating EFSMs (CEFSM) over $Act$ is a parallel composition of a finite set of EFSMs $F_1, \ldots, F_n$ for a given synchronization function. A state in the network is a tuple of the form $\langle \langle l_1, \sigma_1 \rangle, \ldots, \langle l_n, \sigma_n \rangle \rangle$, where $\langle l_i, \sigma_i \rangle$ is a state of $F_i$. We assume a hand-shaking synchronization function similar to that of CCS [26]. A transition of a CEFSM is then either (i) an internal transition of one EFSM, i.e., $\langle \langle l_1, \sigma_1 \rangle, \ldots, \langle l_k, \sigma_k \rangle, \ldots, \langle l_n, \sigma_n \rangle \rangle \xrightarrow{\tau} \langle \langle l'_1, \sigma'_1 \rangle, \ldots, \langle l'_k, \sigma'_k \rangle, \ldots, \langle l'_n, \sigma'_n \rangle \rangle$ if $\langle l_k, \sigma_k \rangle \xrightarrow{\tau} \langle l'_k, \sigma'_k \rangle$ or (ii) synchronization of EFSMs, i.e., $\langle \langle l_1, \sigma_1 \rangle, \ldots, \langle l_k, \sigma_k \rangle, \ldots, \langle l_m, \sigma_m \rangle, \ldots, \langle l_n, \sigma_n \rangle \rangle \xrightarrow{\alpha} \langle \langle l'_1, \sigma'_1 \rangle, \ldots, \langle l'_k, \sigma'_k \rangle, \ldots, \langle l'_m, \sigma'_m \rangle, \ldots, \langle l'_n, \sigma'_n \rangle \rangle$ if $\langle l_k, \sigma_k \rangle \xrightarrow{\alpha} \langle l'_k, \sigma'_k \rangle$, $\langle l_m, \sigma_m \rangle \xrightarrow{\alpha'} \langle l'_m, \sigma'_m \rangle$, and $\alpha'$ and $\alpha'$ are complementary synchronization actions.

Wherever it is clear from the context, we will use term model state denoted $s$ to refer to a state of a CEFSM and the term model transitions denoted $s \xrightarrow{\tau} s'$ or $t$ for a CEFSM transition.
2.2 Test Case Generation

We will focus the presentation on generating test suites with a certain coverage in a CEFSM. Coverage criteria are often used by testing engineers to specify how thorough a test suite should test a system under test. Examples of coverage criteria used in model-based testing include structural criteria such as location or edge coverage, data-flow properties such as definition-use pair coverage, and semantic coverage on, e.g., states of an EFSM or the time regions of a timed automata [28,30]. A coverage criterion typically consists of a list of items to be covered or reached. We shall call those items coverage items, and use \( C \) to denote a set of coverage items, \( C_0 \) the initial \( C \), and \(|C|\) to denote the size of \( C \).

If the coverage criterion stipulates a path property of the kind used in, e.g., data flow criteria as definition-use pairs, we need to handle information about partially satisfied coverage items. We use the definition-use pair coverage criterion [10] to illustrate the concept. It should cover all paths where a given variable \( x \) is first defined in an EFSM edge \( e_d \) active in a transition \( t_d \), and later used in (usually) another EFSM edge \( e_u \) active in a transition \( t_u \), without any redefinitions of \( x \) along the path from \( t_d \) to \( t_u \). We shall store such partial coverage item, i.e., that \( x \) was defined on the EFSM edge \( e_d \) active in \( t_d \), in a set denoted \( A \).

In the algorithms, we shall extend the CEFSM state \( s \) to \((s, C, A)\) or \((s, C)\) when \( A \) is not needed. We further extend the model transition relation to transitions of the form \((s, A, C) \xrightarrow{t} (s', A', C')\) where \( t \) is a model transition \( s \xrightarrow{\alpha} s' \), \( C' \) and \( A' \) are the result of updating \( C \) and \( A \) according to the coverage achieved in transition \( s \xrightarrow{\alpha} s' \). For a detailed description of how \( A' \) and \( C' \) are updated, see e.g., [14].

We shall use traces to represent test cases generated from models. We use \( \epsilon \) to denote the empty trace, and \( \omega.t \) to denote the trace \( \omega \) extended with transition \( t \). Further, we use \(|\omega|\) to denote the length of \( \omega \), defined as \(|\epsilon| = 0\) and \(|\omega.t| = |\omega| + 1\).

3 A Model Checking Approach to Test Suite Generation

3.1 A Local Algorithm

The problem of generating a test suite for a given coverage criteria by reachability analysis has been studied in many settings in the literature, see e.g., [25,19,13,4]. The authors of this paper suggest an algorithm for minimal test suite generation from models of real-time systems described as networks of timed automata in [13] and for untimed systems modeled as extended finite state machines in [4]. A version of these algorithms is shown in Figure 1, but modified so that it returns a shortest path (in the number of steps) with maximum coverage, if the algorithm is executed in a breadth-first manner.

The algorithm is essentially an ordinary reachability analysis algorithm that uses two data structures \( \text{WAIT} \) and \( \text{PASS} \) to hold states waiting to be examined and states already examined, respectively. In addition, the global integer variable \( \text{max} \) is used to (invariantly) store the maximum coverage witnessed so far, and the variable \( \omega^{\text{max}} \) stores a path reaching a state with maximum coverage. Initially \( \text{PASS} \) is empty and \( \text{WAIT} \) holds the initial combined state of the form \((s_0, C_0, \epsilon)\), where \( s_0 \) is the initial state of the model, \( C_0 \) is the coverage of \( s_0 \), and \( \epsilon \) is the empty path.
\( \text{(01) } \text{PASS} := \emptyset \); \text{WAIT} := \{(s_0, C_0, \epsilon)\}; \omega_{\text{max}} := \epsilon; \text{max} := |C_0| \)

\( \text{(02) } \text{while } \text{WAIT} \neq \emptyset \text{ do} \)

\( \text{(03) } \text{select } (s, C, \omega) \text{ from } \text{WAIT} \text{; add } (s, C, \omega) \text{ to } \text{PASS} \)

\( \text{(04) for all } (s', C', \omega.t) : (s, C, \omega) \xrightarrow{\epsilon} (s', C', \omega.t) \text{ do} \)

\( \text{(05) if } |C'| > \text{max} \text{ then} \)

\( \text{(06) } \omega_{\text{max}} := \omega.t; \text{max} := |C'| \)

\( \text{(07) if } \neg \exists (s_i, C_i, \omega_i) : (s_i, C_i, \omega_i) \in \text{PASS} \cup \text{WAIT} \land s_i = s' \land C_i = C' \text{ then} \)

\( \text{(08) add } (s', C', \omega.t) \text{ to } \text{WAIT} \)

\( \text{(09) od} \)

\( \text{(10) od} \)

\( \text{(11) return } \omega_{\text{max}} \)

Fig. 1. A reachability analysis algorithm for test suite generation.

The lines (03) to (08) are repeated until \text{WAIT} is empty. Alternatively, if the maximal number of coverage items is known beforehand, the loop can terminate when the coverage is reached. At line (03) a state is taken from \text{WAIT}, and at line (04) the successors of the state are generated. At line (05) and (06) a new path is saved and a new maximum coverage is saved if the current successor covers more items than the previous maxima. The successor state \((s', C', \omega.t)\) is put on \text{WAIT} if there is no state with the same model state and the same set of covered items, i.e., no state \((s_i, C_i, \omega_i)\) with \(s_i = s'\) and \(C_i = C'\) can be found in \text{WAIT} or \text{PASS}.

It can be shown (see e.g., [13,4]) that the algorithm of Figure 1 returns a shortest path with maximum coverage if the select in line (03) is done so that the algorithm explores the state space in breadth-first order.

**Resets:** Note that the algorithm of Figure 1 may return a trace \(\omega_{\text{max}}\) that does not include all feasible coverage items. This can happen if there are two states \(s_i\) and \(s_j\) in the state space of the model, such that \(s_j\) cannot reach \(s_i\) or the other way around. We can avoid this problem by adding a state \((s_0, C, \text{reset})\) to every successor set at line (04), where \text{reset} is a distinct symbol representing that the model restarts from its initial state. This guarantees that the algorithm in Figure 1 will always return a path with all feasible coverage.

**Coverage subsumption:** A first improvement of the algorithm, described in [14] and in analogy with the inclusion abstraction described in [7], is to change line (07) so that instead of requiring equality of the coverage items \(C_i = C'\), inclusion of coverage items is used, i.e., \(C' \subseteq C_i\).\ The algorithm will now prune an extended state (and thus the part of the state space it may reach) if there exists an extended state with the same model state and a (non-strict) superset of its coverage.

It is also possible to further improve the algorithm in the dual case, i.e., if a state \((s', C', \omega.t)\) is put on \text{WAIT}, such that states \((s_i, C_i, \omega_i)\) exist in \text{WAIT} or \text{PASS} with \(s' = s_i\) and \(C' \supset C_i\). In this case, all states \((s_i, C_i, \omega_i)\) can be removed from \text{WAIT} and \text{PASS}. Note that, as a consequence some states put on \text{WAIT} will never be further explored. Instead subsuming states will be explored. This in turn may change the order in which
states are searched. The same technique has successfully been used to speed up model-checking tools such as UPPAAL [2]. The result is an algorithm that explores fewer states, but ordinary breadth-first search is no longer guaranteed to produce a shortest trace.

3.2 Partial Coverage

The algorithm in Figure 1 is applicable to coverage items (i.e., criteria) that can be determined locally in a single model transition, such as the location or edge coverage criteria. If the coverage criterion stipulates a path property of the kind used in e.g. data-flow criteria as definition-use pairs, the algorithm must be adjusted to handle information about partial coverage items.

Algorithms inspired by model-checking for this class of coverage criteria have been proposed in, e.g., [13,14,4,19]. To modify the algorithm of Figure 1 amounts to storing the partial coverage items in the structure $C$, together with the ordinary coverage items, and modify the behavior of the operator $|C|$ (used at line (06)) so that partial coverage items are not considered. That is, partial coverage is represented and stored in the same way as ordinary coverage, but they are not considered when the number of (fully satisfied) coverage items is computed.

We also note that the coverage subsumption discussed above is not affected by the existence of partial coverage items in $C$. The reset must also be done on the partial coverage items, i.e., $(s_0, A_0, C, \omega, reset)$ is added at successor generation.

4 A Global Algorithm for Test Suite Generation

A well-known problem with algorithms like the one described in the previous section is the time consumed to explore the state space, and the space required to represent WAIT and PASS. The algorithm in Figure 1 explores states of the form $(s, C, \omega)$, resulting in a state space with size defined by the number of model states $s$ in product with the number of possible coverage sets $C$ (the trace $\omega$ does not influence the number of states in the state space).

In this section, we describe algorithms that avoid exploring all states of the form $(s, C, \omega)$ and still generates a reasonable test suite. The idea is to collect and combine information from the whole generated state space so that each model state $s$ is not explored more often than necessary. In particular, we store a set $COV$ of all distinct coverage items covered so far, i.e., $COV = \bigcup_{i} C_i$ for all explored states $(s_i, C_i, \omega_i)$.

Additional information, including a trace to each coverage item $c \in COV$ is stored in a structure $SUITE$, that is used to generate the test suite returned by the algorithm. We first describe an algorithm for coverage criteria without partial coverage items in Section 4.1, followed by an algorithm handling partial coverage in Section 4.2.

4.1 A Global Algorithm

The algorithm shown in Figure 2 is a modified version of the algorithm in Figure 1. It works in a similar way, but collects coverage from all explored states (i.e., branches) in the variables $COV$ and $SUITE$. The variable $COV$ holds the total set of coverage
(01) \text{PASS} := \emptyset; \text{WAIT} := \{(s_0, C_0, \epsilon)\}; \text{SUITE} := \emptyset; \text{COV} := C_0$

(02) while \text{WAIT} \neq \emptyset do

(03) select \((s, C, \omega)\) from \text{WAIT}; add \((s, C, \omega)\) to \text{PASS}

(04) for all \((s', C', \omega.t)\) : \((s, C, \omega) \Rightarrow_c (s', C', \omega.t)\) do

(05) if \(C' \not\subseteq \text{COV}\) then

(06) add \((\omega.t, C')\) to \text{SUITE}; \text{COV} := \text{COV} \cup C'

(07) if \neg \exists (s_i, C_i, \omega_i) : (s_i, C_i, \omega_i) \in \text{PASS} \cup \text{WAIT} \land s_i = s' then

(08) add \((s', C', \omega.t)\) to \text{WAIT}

(09) od

(10) od

(11) return \text{SUITE}

Fig. 2. A global coverage algorithm for test suite generation.

items found by the algorithm, i.e., \(\text{COV} = \bigcup_i C_i\) for all explored states \((s_i, C_i, \omega_i)\).

For every explored state with new coverage, a tuple \((\omega_i, C_i)\) is added to the set \(\text{SUITE}\).

This makes \(\text{SUITE}\) set of tuples with one trace \(\omega\) to each coverage item in \(\text{COV}\). With ordinary breadth-first search strategy, \(\text{SUITE}\) will hold a trace to coverage item with the minimum number of model transitions. The additional information stored in \(\text{SUITE}\) will be used to improve the algorithm and the test suite, later in this section.

The loop of the algorithm in Figure 2 has two differences from the algorithm in Figure 1. In lines \(05\) and \(06\) the variables \(\text{COV}\) and \(\text{SUITE}\) are updated if the explored state contains new coverage items that the algorithm has not seen before. Note that in line \(07\), we do not consider the coverage \(C'\) of the generated states. As a result, a state \((s', C', \omega.t)\) is not further explored (i.e., added to \(\text{WAIT}\)) if the model state \(s'\) has been previously explored. The algorithm terminates when \(\text{WAIT}\) is empty which is the case when all reachable states from \(s_0\) have been explored. At this point, each model state has been explored only once, \(\text{COV}\) contains all reachable coverage items, and \(\text{SUITE}\) includes at least one trace to each coverage item in \(\text{COV}\). We shall return to the problem of generating a test suite from \(\text{COV}\) in Section 4.3.

4.2 Partial Coverage

We now describe how to modify the algorithms above so that it can be used for coverage criteria that requires partial coverage items (in analogy with the modified algorithm presented in Section 3.2). Recall that partial coverage items are needed when the coverage criteria requires path properties to be covered, like in the definition-use pair criterion (see Section 2.2).

The modified algorithm is shown in Figure 3. It operates on extended states of the form \((s, A, C, \omega)\), where \(C\) and \(A\) are the coverage items and the partial coverage items respectively, collected on the path \(\omega\) reaching \(s\). The only principal difference compared to the algorithm of Figure 2 is on line \(07\) where the most recently generated state \((s', A', C', \omega.t)\) is examined. Here, the state is not further explored if an already explored state \((s_i, A_i, C_i, \omega_i)\) with \(s_i = s'\) and \(A' \subseteq A_i\) exists in \(\text{PASS}\) or \(\text{WAIT}\). If this is the case, it can be deduced that further exploration of \((s', A', C', \omega.t)\) is not needed.
(01) \( \text{PASS} := \emptyset; \ \text{WAIT} := \{(s_0, A_0, C_0, \epsilon)\}; \ \text{SUITE} := \emptyset; \ \text{COV} := C_0 \)
(02) \( \text{while } \text{WAIT} \neq \emptyset \text{ do} \)
(03) \( \text{select } (s, A, C, \omega) \text{ from } \text{WAIT}; \ \text{add } (s, A, C, \omega) \text{ to } \text{PASS} \)
(04) \( \text{for all } (s', A', C', \omega.t) : (s, A, C, \omega) \in \text{PASS} \text{ do} \)
(05) \( \text{if } C' \not\subseteq \text{COV} \text{ then} \)
(06) \( \text{add } (\omega.t, C) \text{ to } \text{SUITE}; \ \text{COV} := \text{COV} \cup C' \)
(07) \( \text{if } \neg \exists (s_i, A_i, C_i, \omega_i) : (s_i, A_i, C_i, \omega_i) \in \text{PASS} \cup \text{WAIT} \land s_i = s' \land A' \subseteq A_i \text{ then} \)
(08) \( \text{add } (s', A', C', \omega.t) \text{ to } \text{WAIT} \)
(09) \( \text{od} \)
(10) \( \text{od} \)
(11) \( \text{return } \text{SUITE} \)

Fig. 3. A global algorithm with partial coverage items.

since the state is not able to contribute coverage items other than those that further exploration of \((s_i, A_i, C_i, \omega_i)\) will yield.

The algorithm of Figure 3 terminates when \( \text{WAIT} \) it empty. At this point, all reachable model states from \( s_0 \) have been explored. \( \text{COV} \) contains all reachable coverage items, and \( \text{SUITE} \) is a set of pairs of the form \((\omega_i, C_i)\), where \( \omega_i \) is a trace ending in a state with coverage \( C_i \), and \( \bigcup_i C_i = \text{COV} \). It is easy to prove that the algorithm is sound. It is also complete since for each reachable partial coverage item \( a_i \), an extended state \((s, A, C, \omega)\), such that \( a_i \in A \), has been explored. This guarantees that all feasible coverage items will be generated since item \( c_i \in \text{COV} \) depends only on one (or zero) partial coverage items in \( A \).

4.3 Improving the Test Suite

When the algorithm above terminates, \( \text{SUITE} \) is a set \( \{(\omega_0, C_0), \ldots, (\omega_{n-1}, C_{n-1})\} \).
Ideally, this set should be reduced so that the total coverage \( \bigcup_i C_i \) is not changed, and the length of the test suite, i.e., \( \sum_i |\omega_i| \), is minimized. The remaining traces \( \omega_i \) can then be used as the test suite. However, selecting a subset of traces with this property is a version the well-known set covering problem which is NP-hard [21].

The likewise well-known technique of prefix elimination can be used as an approximative solution to the problem, i.e., to make sure that there are no pair of traces \( \omega, \omega' \) in \( \text{SUITE} \) such that \( \omega \) is a prefix of \( \omega' \) of the other way around. However, this approach has some obvious problems, including the fact that \( \text{SUITE} \) could still include redundant traces, i.e. traces that can be removed without reducing the total coverage of \( \text{SUITE} \).

We have chosen an algorithm that can be performed incrementally, as part of the main test case generation algorithm. It can also be applied to \( \text{SUITE} \) when the main algorithm has terminated. It checks that each \((\omega_j, C_j)\) in \( \text{SUITE} \) satisfies the condition \( C_j \not\subseteq C_0 \cup \ldots \cup C_{j-1} \cup C_{j+1} \cup \ldots \cup C_{n-1} \), i.e., \((\omega_j, C_j)\) contributes to the total coverage of \( \text{SUITE} \). As we shall see in the next section, this approach has worked well in our experiments.
Table 3. Time (in seconds) and space (in MB) performance of the algorithms.

<table>
<thead>
<tr>
<th></th>
<th>Local Algorithm</th>
<th></th>
<th></th>
<th></th>
<th>Global Algorithm</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>time mem len</td>
<td>states tr</td>
<td></td>
<td></td>
<td>time mem len</td>
<td>states tr</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Train e 6 (0)</td>
<td>full</td>
<td>2.9 9.6 15</td>
<td>3353 1</td>
<td></td>
<td>1.25 9.3 15</td>
<td>1645 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>reset</td>
<td>3.85 10.5 15</td>
<td>7375 1</td>
<td></td>
<td>- - - -</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Train du 12 (5)</td>
<td>full</td>
<td>37 14 47</td>
<td>27129 1</td>
<td></td>
<td>2.4 9.3 156</td>
<td>1717 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>reset</td>
<td>107 33 47</td>
<td>114697 1</td>
<td></td>
<td>- - - -</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Philips du 109 (42)</td>
<td>stop 30</td>
<td>9 10 30</td>
<td>5797 1</td>
<td></td>
<td>1 9.2 80 170</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>stop 60</td>
<td>1085 87.7 69</td>
<td>461305 1</td>
<td></td>
<td>2.6 9.2 204</td>
<td>393 16</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>full</td>
<td>- - - -</td>
<td>- - - -</td>
<td></td>
<td>16.4 10.6 681</td>
<td>7579 37</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WAP e 68 (0)</td>
<td>stop 20</td>
<td>4.29 12.7 35</td>
<td>5105 1</td>
<td></td>
<td>4.17 12.5 161</td>
<td>4361 5</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>stop 30</td>
<td>30.8 31.2 86</td>
<td>35615 1</td>
<td></td>
<td>4.51 12.5 175</td>
<td>4395 5</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>stop 68</td>
<td>- - - -</td>
<td>(7348332) 0</td>
<td></td>
<td>38 32 818</td>
<td>37503 12</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>full</td>
<td>- - - -</td>
<td>- - - -</td>
<td></td>
<td>3871 1946 818</td>
<td>3850877 12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WAP ei 68 (68)</td>
<td>stop 20</td>
<td>10.85 12.7 58</td>
<td>5404 1</td>
<td></td>
<td>8 12 58 5299</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>stop 30</td>
<td>525 52 93</td>
<td>71328 1</td>
<td></td>
<td>15.9 14.9 189</td>
<td>7291 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>stop 68</td>
<td>- - - -</td>
<td>- - - -</td>
<td></td>
<td>280 111.2 1718</td>
<td>200011 19</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>full</td>
<td>- - - -</td>
<td>- - - -</td>
<td></td>
<td>4093 2025 1718</td>
<td>3937243 19</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

5 Experiments

For the experiments in this section we use our tool UPPAAL COVER that takes as input a timed automata model and a coverage criterion.

Models and Coverage Criteria: We will use three models that are documented in the literature: a train gate example (Train) where four trains and a controller are modeled [32], a audio-control protocol with bus collision detection by Philips (Philips) [3], and a WAP stack (WAP) modeled and tested in [15].

We present experiments of three different coverage criteria edge (e), definition-use pair (du), and edge init (ei). In the edge coverage criterion a coverage item is a traversed edge in an automaton. If several instances of the same automaton are used, it does not distinguish between different instances exercising the edge. The du-pair criterion is described in Section 2.2 of this paper. The edge init coverage criterion requires an edge to be exercised, as in the edge coverage criterion. For the coverage item to be satisfied, the test case must also put the system back to a state that is similar to the initial system state.

Results: In Table 3 the performance of the local and global algorithms is presented. The algorithms were executed using breadth-first search strategy on a SUN Ultra SPARC-II 450MHz.

The leftmost column of the table specifies the input used in the experiments, i.e., the model, the coverage criterion, the number of coverage items, and (in parentheses) the number of partial coverage items existing in the model. In the second column the following keywords are used: full for exhaustive search, stop x for termination after x
found coverage items, and reset if resets are used in the model (as described in Section 3.1 this is applicable only in the local algorithm). For both the local and global algorithm we give the following numbers/columns, generation time (time), memory (mem), length of the trace (len), number of states generated (states), and number of traces generated (tr).

The rows marked Train e 6 (0) show performance for the train gate model with the edge coverage criterion on the instances of the train automaton. There are six coverage items to be covered, with zero partial coverage items. The global algorithm generates 1645 states which is the size of the model. The local algorithm generates 3353 or 7375 states without and with resets, respectively.

For the rows Train du 12 (5) the definition-use criterion has been used. There are 12 different coverage criteria and five partial coverage items. The size of the generated state space of the global algorithm increases (due to the partial coverage items) modestly to 1717 (+4.3% compared with the actual size of the model state space). For the local algorithm this increase is from 3357 states to 27129 (or 114697 when resets are used). We note that the global algorithm performs substantially better than the local algorithms. In fact, it generates only 6% (or 2%) of the states used by the local algorithm(s). The gain in execution time is similar.

For the models in the rest of the table we have not been able to run exhaustive analysis with the local algorithm, nor have been able to use resets. Still the experimental results show how the algorithms scale up for different models and coverage criteria. In all the examples, the global algorithms outperforms the local algorithm.

6 Conclusion

In this paper, we have studied algorithms and ideas for test suite generation applicable to automata models with semantics defined as a (finite) transition system. We have reviewed algorithms derived from ordinary reachability analysis algorithm, similar to those used in many model-checking and planning tools. Such algorithms can be used to generate test suites that follow a given coverage criterion and are optimal in e.g. the total number of model transitions the test suite will exercise. We have further elaborated these algorithms by adopting existing abstraction and pruning techniques often used in model-checking algorithms.

The main contribution of this paper, is a novel global algorithm for model-based generation of test suites following a given coverage criteria. At any given point, the algorithm — which is inspired by the priorly described reachability analysis algorithms — uses knowledge about the total coverage found in the currently generated state space to guide and prune the remaining exploration. In this way, the algorithm avoids unnecessary exploration and generates a test suite with reasonable characteristics.

All algorithms presented in this paper have been implemented in our test case generation tool UPPAAL COVER. To compare and evaluate the algorithms, we have performed a number of experiments on a set of models previously described in the literature. In particular, the evaluation gives experimental evidence that the suggested global algorithm uses substantially less memory and time than local algorithms, and outputs test suites
that are not far from optimal. In this respect, the suggested global algorithm increases the maximum size of models for which test suites can be algorithmically generated.

References


Generating Scenarios by Multi-Object Checking

Maik Kollmann\textsuperscript{1} and Yuen Man Hon\textsuperscript{2}

\textsuperscript{1} Technical University Braunschweig – Institute of Information Systems, Mühlenpfordtstraße 23, 38106 Braunschweig, Germany
\texttt{M.Kollmann@tu-bs.de}
\textsuperscript{2} Technical University Braunschweig – Institute for Railway Systems Engineering and Traffic Safety, Pockelsstraße 3, 38106 Braunschweig, Germany.
\texttt{Y.Hon@tu-bs.de}

\textbf{Abstract.} These days, many systems are developed applying various UML notations to represent the structure and behavior of (technical) systems. In addition, for safety critical systems like Railway Interlocking Systems (RIS) the fulfillment of safety requirements is demanded. UML-based Railway Interlocking (UML-based RI) is proposed as a methodology in designing and developing RIS. It consists of infrastructure objects and UML is used to model the system behavior. This design is validated and demonstrated by using simulation with Rhapsody. Automated verification techniques like model checking have become a standard for proving the correctness of state-based systems. Unfortunately, one major problem of model checking is the state space explosion if too many objects have to be taken into account. Multi-object checking circumvents the state space explosion by checking one object at a time. We present an approach to enhance multi-object checking by generating counterexamples in a sequence diagram fashion providing scenarios for model-based testing.

\textbf{Key words:} Scenario Generation, Counterexample Generation, State Modeling, Multi-Object Checking, UML-based Railway Interlocking System

\section{Introduction}

Nowadays, during different phases of system development, Unified Modeling Language (UML) notations [OMG05] is used as the modeling tool to specify the structure and behavior of (technical) systems. These systems contain components or objects that interact via channels or a bus in order to provide a specified functionality. In addition, for safety critical systems like Railway Interlocking Systems (RIS) the fulfillment of safety requirements is demanded. The proof that a safety critical system provides safe operation determines the conformance to the desired safety integrity level (SIL) of such a system [CEN06,Bel05]. In general, safety critical systems are demanded to reach SIL 4, the highest level. According to EN 50128 [CEN06], SIL 4 RIS highly recommend a proof that the safety requirements are fulfilled up to a tolerable level by the RIS software.
RIS are responsible for establishing safe routes for trains that are scheduled to pass through or stop at a railway station. Safe routes ensure that trains cannot be driven into tracks that are occupied or may become occupied by other trains. A safe route ensures the proper settings of infrastructure elements like points and signals along the route. These elements can be modeled using UML class diagrams and UML state machines [HK06,BH06]. The next step is to prove that a model of a concrete interlocking is working correctly and that it conforms the safety requirements. Such requirements are the absence of conflicting routes, etc. Automated verification techniques like model checking have become a standard for proving the correctness of state-based systems. Unfortunately, model checking suffers from the state space explosion problem if too many objects have to be taken into account.

Multi-object checking [EKP03] circumvents the state space explosion by its nature checking one object at a time. Multi-object checking relies on the sound and complete multi-object logics $D_1$ and $D_0$ [EC00] that are based on well-known logics like CTL or LTL. $D_1$ is supposed to be more intuitive for specification, as the interaction among different objects can be specified. However, it cannot be verified directly applying model checking. Formulas in $D_1$ can be translated into an equivalent set of $D_0$ formulas. Each checking condition is bound to a particular object in a $D_0$ formula. As a result, $D_0$ formulas can be verified automatically by model checking.

There are similar properties between multi-object checking and traditional model checking. They provide an automatic proof that a certain (safety) property holds. In addition, model checking provides the opportunity to analyze a system model if the checking condition does not hold. If a model violates a checking condition, multi-object checking provides a counterexample from a model checking tool for the outermost object. Taking the mentioned communication with other objects into account, this counterexample trace had to be analyzed manually. We propose an observer-based approach to synthesize a multi-object counterexample that is closed with respect to interaction of objects.

Model checking tools like SPIN [Hol97] or SMV/NuSMV [McM96,CCB+02] incorporate the ability to illustrate that a model does not satisfy a checking condition with a textual, a tabular or a sequence chart-like representation of the involved states. We believe that a graphical representation like message sequence charts is the most convenient one from a users point of view to illustrate a counterexample in a multi-object system. Beyond providing a graphical representation for certain scenarios, sequence charts have been successfully applied to generate test cases [Ebn05]. In this contribution, we present an approach for generating test cases automatically based on the system specification instead of deriving them manually.

2 Railway Interlocking Systems

RIS are often designed specifically according to the layouts of stations. When the layout of a station is changed, the corresponding RIS has to be modified.
This causes high costs for resources and time \cite{vDFK+98}. In order to reduce the effort in modifying RIS, one can develop RIS under the object oriented approach. With this approach, it is only necessary to specify the new neighboring structure of the objects when the layout of the station is changed instead of modifying the whole interlocking system completely.

The geographical interlocking is used as the internal logic of the interlocking system that consists of infrastructure elements as objects. The objects interact among each other with events and actions to develop safe routes. Objects, events and actions can be captured by using state modeling. In \cite{BFG04} StateMate \cite{HLN02} is proposed and applied to model the functional requirements of interlocking systems. In contrast, in this work, UML state machines are used for state modeling. RIS that are developed by applying the geographical interlocking and using UML as specification tool are called UML-based Railway Interlockings (UML-based RI).

Figure 1 shows the railway station CStadt that is used to check the feasibility of developing a UML-based RI. There are different kinds of infrastructure elements that are located within this station: tracks, points and signals.

Fig. 1. Track layout of station CStadt

Tracks are the basic infrastructure elements that trains move on. The authorized movement of railway vehicles on the tracks can be classified into train movements and shunting movements. Train movements are only allowed on main tracks, while making up a train, called shunting movements, are undergone only on sidings \cite{Pac04}. In figure 2, main tracks are \textit{GA}$_1$, \textit{GA}$_2$, \textit{GA}$_3$ and \textit{GA}$_4$. Only train movements are analyzed in the current model of UML-based RI. Instead of moving only straight ahead, trains can also turn to another track via turnouts or crossing. Turnouts and crossing are composed of points. Points (e.g., \textit{W}$_1$) can be set to right and left position. In order to ensure the train can be driven into the current track, RIS need to ensure the correct position of points.

Along the track, one can find another kind of infrastructure component: the signal. Signals (e.g., \textit{P}$_1$) can be used to control train movements by showing the aspects. Two types of aspects will be considered in this work, they are clear and stop aspect. Some signals can also indicate the allowed speed of the corresponding track section. Indication of speed will not be considered in the current work.

As mentioned before, the main task of RIS is to develop a safe route for the approaching train. There are two requirements of a safe route. First, the infrastructure elements along the route have been properly set. For example, the points are set in the correct positions and no other trains can be driven into the safe route from the divergent direction. Flank protection is used to prevent other
trains to drive into a safe route via points. Furthermore, no conflicting routes are issued at the same time as the safe route. This means the infrastructure elements which belong to the route can only be used exclusively by one train. If the requested route of the approaching train fulfills those requirements, in other words, it is a safe route, then this route and the infrastructure elements along this route will be locked exclusively for this train. No other train can use the same route, so that collisions can be avoided. The mentioned safety requirements can be ensured by executing two procedures: checking the availability of infrastructure elements and providing flank protection for the route.

3 Multi-Object Systems and Multi-Object Checking

One of the possible solutions in handling the state space explosion is to develop a method, such that a condition can be verified without building up the complete state space and only those state variables of objects that are involved in the condition are considered during the verification. In this approach, the correctness of a formula which involves a single object can be verified by checking the state space of this object locally and this type of formula can be called a local condition. When there are more objects defined, a formula is called a global condition. This global condition can be broken down into a set of local conditions of each object and communications between objects. The correctness of the global condition can be ensured by checking those local conditions and the existence of communications among the objects. Multi-object checking is a method that comprises the above ideas. It can be used to verify state-based multi-object system efficiently without building the complete state space. In multi-object checking, the system consists of objects and objects communicate among each other synchronously in an RPC-like fashion. The communication between objects must be specified before the verification. Each of the objects has a signature that describes its properties, for example, its attributes, actions and valid propositions. This signature can for example be captured in an Finite State Machine (FSM).

Let $I$ be a finite set of identities representing sequential objects. Misusing terminology slightly, we speak of object $i$ when we mean the object with identity $i$. Each object $i \in I$ has an individual set $P_i$ of atomic state predicate symbols,
its signature. In applications, these predicate symbols may express the values of attributes, which actions are enabled, which actions have occurred, etc.

We use the multi-object logic $D_1$ (cf. figure 3) as described in [EC00,ECSD98] with a local logic $i.LTL_i$ over signature $P_i$ for each object $i \in I$. $D_1$ allows to use formulas $j:LTL_j$, $i:LTL_i$, $j?LTL_j$ etc. for any other object $j \in I$ as subformulas within $i.LTL_i$. These constituents are called communication subformulas.

$$D_1 ::= i.LTL_i$$

$$F_{i,LTL} ::= i.LTL_i \mid i?LTL_i \mid i!LTL_i \mid i^\ast LTL_i \mid \neg i.LTL_i \mid i.LTL_i \land i.DTL_i \mid i.DTL_i \lor i.LTL_i$$

$$L_{i,LTL} ::= ⊥ \mid P_i \mid \neg L_{i,LTL} \mid L_{i,LTL} \land L_{i,LTL} \mid L_{i,LTL} \lor L_{i,LTL} \mid L_{i,LTL} \Rightarrow L_{i,LTL} \mid L_{i,LTL} \Leftrightarrow L_{i,LTL} \mid X L_{i,LTL} \mid F L_{i,LTL} \mid G L_{i,LTL} \mid L_{i,LTL} \cup L_{i,LTL} \mid L_{i,LTL} \mid L_{i,LTL} \mid L_{i,LTL} \mid L_{i,LTL} \mid L_{i,LTL} \mid L_{i,LTL} \mid L_{i,LTL} \mid L_{i,LTL} \mid L_{i,LTL} \mid L_{i,LTL} \mid L_{i,LTL}$$

$$C_{i,LTL} ::= \ldots \mid F_{j,LTL} \mid \ldots \quad (j \in J, J \equiv I \setminus \{i\})$$

**Fig. 3.** Syntax definition of Multi-Object Logic $D_1$ (with local logic LTL)

The global conditions that involve more than one object are specified by the multi-object logic $D_1$. Logic $D_1$ allows one to specify temporal formulas or propositions that an object $i$ fulfills locally. It also allows one to specify temporal formula or propositions that other objects satisfy for an object if there is synchronized communication between them. For example, a formula $o_1.\neg EF(\text{state} = \text{critical} \land o_2.\text{(state} = \text{critical}))$ means that there exists a synchronized communication currently between objects $o_1$ and $o_2$, such that $o_2$ guarantees for $o_1$ that $o_2$ is in the critical state (cf. figure 4). $o_2.\text{(state} = \text{critical})$ is called the communication sub-formula in this method.

**Fig. 4.** Objects $o_1$ and $o_2$ must not reside in the critical section together: $o_1.\neg EF(\text{state} = \text{critical} \land o_2.\text{(state} = \text{critical}))$

There is a sublogic of $D_1$ called $D_0$ (cf. figure 5). Formulas that involve communication with different objects cannot be expressed in $D_0$ directly. However,
the synchronized communication between two objects can be explicitly specified with the properties of the objects in $D_0$.

$$
D_0^1 ::= i.i_{LTL}^i | i.C_0
$$

$$
L_0^\text{LTL} ::= \bot | P_i | \neg L_0^\text{LTL} | L_0^\text{LTL} \land L_0^\text{LTL} | L_0^\text{LTL} \lor L_0^\text{LTL} |
$$

$$
\Rightarrow L_0^\text{LTL} \Rightarrow L_0^\text{LTL} \Leftrightarrow L_0^\text{LTL} |
$$

$$
X L_0^\text{LTL} | F L_0^\text{LTL} | G L_0^\text{LTL} | P_0^\text{LTL} U L_0^\text{LTL} |
$$

$$
Y L_0^\text{LTL} | O L_0^\text{LTL} | H L_0^\text{LTL} | L_0^\text{LTL} S L_0^\text{LTL} |
$$

$$
@J | @!J | @?J
$$

$$
C_0^i ::= ... | (P_i \Rightarrow j.P_j) | ... \quad (j \in J , J \equiv I \setminus \{i\})
$$

Fig. 5. Syntax definition of Multi-Object Logic $D_0$ (with local logic LTL)

[EC00] presents a transformation to break global $D_1$ checking conditions down into sets of $D_0$ conditions and communication symbols (cf. figure 6). These symbols have to be matched with existing ones according to the communication requirements. In addition, $D_0$ conditions can be verified locally. Informally, the communication symbols are determined inside out. A $D_1$ formula $\psi$ holds iff the outermost $D_0$ formula $\psi'$ holds. This is elaborated in [KH07,EKP03].

$$
\psi \equiv i.(... j.(\varphi) ...)
$$

$$
\psi' \equiv i.(... q.j ...) \quad \varphi' \equiv j.(q;i \Rightarrow \varphi)
$$

Fig. 6. $D_1$ to $D_0$ transformation

### 3.1 Model Checking

Multi-object checking has been demonstrated successfully using model checking to determine the communication symbols and to match with existing ones according to the communication requirements in [EKP03]. Model checking consists in verifying a system against a checking condition, also called a formula ($M \models \varphi$, $M$ is the model and $\varphi$ is the formula).

The system that needs to be verified is specified as a (labeled) transition system $T$ and is called the model $M$. A transition system $T = (S, s_0, L, \rightarrow)$ consists of a set of states $S$ ($s_0 \in S$ is the initial state) with labeled transitions ($L$ set of labels; $\rightarrow \subseteq S \times L \times S$ is a set of labeled transitions) [WN95].

Any checking condition that the system needs to satisfy is defined in temporal logic [HR00,CGP00] and is called formula $\varphi$. In model checking, temporal logic
is used to describe the properties of the system over time. Temporal logic models time as a sequence of states. As a result, one can use temporal logic to describe conditions that have to hold as the system evolves [HR00]. Unfortunately, model checking suffers from the state space explosion problem [Val98] if too many objects have to be taken into account during the process of verification.

A widely used feature of model checking is the ability to generate counterexamples if a checking condition is evaluated to \textbf{false}. Each counterexample consists of a sequence of states illustrating the inconformity. The quality of counterexamples has been discussed in [GV03].

As no model checking tool can determine which sequence of states is suitable to illustrate the error if different sequences may be chosen, estimating the quality of a specific counterexample is based on the view of the user. However, little progress has been achieved in this area [ELL01]. The representation of counterexamples varies among the model checking tools. Most of them offer a textual representation of the sequence of states whereas other tools make use of tables or sequence diagrams.

3.2 Multi-Object Checking Counterexamples

As mentioned above, in multi-object checking, a $D_1$ formula holds iff the outermost $D_0$ formula holds. Otherwise, a counterexample has to be generated if this $D_0$ formula does not hold. If model checking is used to verify the outermost $D_0$ formula, an initial part of a counterexample is generated. This forms a part of a global counterexample and it is called a local counterexample. If the initial local counterexample does not contain communication with further objects, it is the global multi-object checking counterexample. Similarly, whenever the local counterexample contains communication with further objects, the local counterexample is a part of a global counterexample.

We concentrate on the class of multi-object checking counterexamples in which communication among objects exists. The initial state of all transition systems does not contain any communication. Consequently, the initial local counterexample refers to model checking counterexamples of finite or infinite length. The order of communication between the objects’s communication partners has to be preserved. The algorithm is executed as follows:

- Apply model checking for each object which is identified as a communication partner of the object under investigation.
- Check whether the communication can be preserved for each of the objects.
- Whenever a further communication partner is discovered iterate the algorithm until all communication is covered.

This algorithm works on the grounds that each communication partner provides a corresponding communication scheme. As it has been mentioned before, it is difficult to determine the expressiveness of a counterexample, the selection of a random counterexample for each object does not guarantee that a global counterexample can be generated. Whenever a communication partner of an object does not provide a demanded communication scheme one of the possibilities
is that, the counterexample of this object has not been properly chosen. This issue can be solved by automatically selecting a different counterexample for the initiating object that shows a different communication scheme. Alternatively, users can interactively examine the part of the global counterexample that has been generated so far.

Both solutions show drawbacks. The first solution implements a kind of greedy/backtracking algorithm with all the well-known (dis-)advantages. The more often local counterexamples have to be regenerated, the less efficient the algorithm is. Finally, the global counterexample may not illustrate the error that clear as expected. In contrast, applying the second strategy, the user may not have enough information to decide whether the partial global counterexample shows the origin of the error. These observations constitute the following strategy for verification to generate counterexamples:

– Depending on a user-specified timeout, we try to generate different local counterexamples.
– If time runs out, the user can judge whether the actual counterexample is useful.
– If the user cannot take any advantage of the given counterexample, he can guide the algorithm by providing domain knowledge.

3.3 Application of the Counterexample Generation Algorithm to UML-based RI

We have introduced a UML-based RI in Section 2. Checking conditions (e.g., if an infrastructure object $o$ is not in use ($state = unused$), it is always possible to establish a route ($lock = route$) using this object: $F := o.(G (state = unused) \Rightarrow F (lock = route))$) that the model is expected to fulfill need to be defined. In [HK06], similar requirements have been checked by simulation, model checking and multi-object checking.

The algorithm of generating counterexamples that has been mentioned can be demonstrated by checking condition $F$. As checking condition $F$ does not hold initially, we have easily discovered the erroneous situation: $GA3$ may not be in use but it may receive a route request from both neighbors $W3$ and $W1$. In

![Fig. 7. Rejection of two concurrent route requests](image-url)
this case the safe (but not functional) response is to reject both route requests \( rr \). We have applied the counterexample generation algorithm and created two observers for objects \( W_3 \) and \( W_1 \) in the first step (cf. figure 7).

![Fig. 8. Creation of two observers for objects \( W_3 \) and \( W_1 \)](image)

The observer for \( W_3 \) monitors the behavior of \( W_3 \) such that a transition \( t' \in T' \) in the observer automaton is a projection of those transitions \( t \in T \) in \( W_3 \) that have the same communication. A state in the observer automaton represents states and transitions that occur between the transitions \( t_i \) and \( t_{i+1} \) in the automaton of \( W_3 \). Checking whether \( W_3 \) can communicate with \( GA_3 \) as expected can be evaluated by checking if the behavior of the observer automaton is possible (cf. figure 8). \( W_3 \) and its observer synchronize on the transitions specified in the observer.

The generated sequence chart in figure 9 presents a global scenario out of the interlocking’s perspective. It has been generated in the same way as described above.

![Fig. 9. Conflicting routes may be established one after the other](image)

### 3.4 Generation of Test Cases by Multi-Object Checking

Test cases are typically derived manually from a given specification. This derivation models a part of the system. These test cases can also be used to model the developed system’s behavior. Both, the system design model and the test model may contain errors. Many checking conditions \( \phi \), given by a (formal) specification, define implications that some conclusion \( \omega \) holds under some premises \( \lambda \) [PPKE07]. Most of the premises and conclusions define a certain state in the system and constitute checking conditions which share the following structure:
G((λ₁ ∧ λ₂ ∧ ... ∧ λₙ) ⇒ Fω). As each λᵢ, 1 ≤ x ≤ n belongs to a particular object i, a D₁ formula that is compliant to the communication requirements among objects can be formulated. A formula φ\text{\textit{example}} := G((λ\text{shared track element} ∧ λ\text{route}_1 ∧ λ\text{route}_2) ⇒ F(ω\text{shared track element})) is translated as follows:

φ\text{\textit{D₁ example}} := \text{shared track element.}(G((λ\text{shared track element} ∧ \text{route}_1:(λ\text{route}_1) ∧ \text{route}_2:(λ\text{route}_2)) ⇒ F(ω\text{shared track element})))

Let all such checking conditions φ evaluate to TRUE by verifying the system model beforehand. We automatically derive checking conditions φ\text{\textit{test}} by negating the conclusion: G(λ₁ ∧ λ₂ ∧ ... ∧ λₙ ⇒ ¬Fω). The negation is applied to the D₁ formulas as well. We derive the following checking condition to generate a test case for checking condition φ\text{\textit{D₁ example}}:

φ\text{\textit{D₁ test}} := \text{shared track element.}(G((λ\text{shared track element} ∧ \text{route}_1:(λ\text{route}_1) ∧ \text{route}_2:(λ\text{route}_2)) ⇒ ¬F(ω\text{shared track element})))

Such checking conditions obviously evaluate to FALSE by multi-object checking and suitable counterexamples are generated applying the greedy/backtracking algorithm (cf. Section 3.2). They illustrate possible evolutions of the system from the initial state to the state in which the condition (¬Fω) does not hold. These evolutions can be used to specify test cases automatically if a mapping of states in the design model to the implementation is given.

In [Ebn05], a translation of Message Sequence Charts into Testing and Test Control Notation Version 3 (TTCN-3) test cases is given. In combination with the generated sequence chart for a checking condition φ\text{\textit{D₁ test}} and an appropriate mapping to the implementation, a TTCN-3 test case can be derived easily.

4 Conclusion

In this contribution, a strategy for generating counterexamples for multi-object checking is described. We have demonstrated the usefulness of our strategy by a case study featuring a UML-based RI. RIS are considered as safety critical systems. The guarantee of the correct behavior throughout the system lifecycle is demanded. In order to save resources in developing and modifying RIS for amended railway layouts, an object oriented approach for establishing interlocking systems is investigated in this work. Infrastructure elements are considered as infrastructure objects in a UML-based RI. The objects of a RIS cooperate with each other to function as an interlocking system.

Multi-object checking has been successfully applied to verify UML-based RIS. We concentrate on verifying the safety aspect of the model. The provided graphical counterexample in this contribution helped correcting the state machines displaying the obviously unhandled situation. We believe that this methodology that improves the understanding and communication among professions of different disciplines, can improve the whole development process of a system.
We have also shown a further step of improving system development by designing a more comprehensible verification strategy. It provides illustrative counterexamples and generates test cases automatically. Therefore, we have demonstrated how TTCN-3 test cases can be derived from checking conditions during the early stage of system development.

The positive feedback from railway engineers who are familiar with sequence chart notations suggests the further development of the user interface of our tool. Future work will enhance the counterexample generation process by the concurrent determination of multi-object checking communication symbols and reusing interim results during the verification process.

References


A Case Study in Matching Test
and Proof Coverage

Lydie du Bousquet, Yves Ledru, Frédéric Dadeau, and Faryel Allouti

Laboratoire Informatique de Grenoble (LIG)
B.P. 72 - F-38402 - Saint Martin d’Hères Cedex - France

Abstract. This paper studies the complementarity of test and deductive proof processes for Java programs specified in JML (Java Modeling Language). The proof of a program may be long and difficult, especially when automatic provers give up. When a theorem is not automatically proved, there are two possibilities: either the theorem is correct and there are not enough pieces of information to deal with the proof, or the theorem is incorrect. In order to discriminate between those two alternatives, testing techniques can be used. Here, we present experiments around the use of the JACK tool to prove Java programs annotated with JML assertions. When JACK fails to decide proof obligations, we use a combinatorial testing tool, TOBIAS, to produce large test suites that exercise the unproved program parts. The key issue is to establish the relevance of the test suite with respect to the unproved proof obligations. Therefore, we use code coverage techniques: our approach takes advantage of the statement orientation of the JACK tool to compare the statements involved in the unproved proof obligations and the statements covered by the test suite. Finally, we ensure our confidence within the test suites, by evaluating them on mutant program killing exercises. These techniques have been put into practice and are illustrated by a simple case study.

Keywords: deductive proof process, combinatorial testing, test suite relevance evaluation, JML

1 Introduction

Software testing has emerged as one of the major techniques to evaluate the conformance between a specification and some implementation. Unfortunately, testing only reveals the presence of errors and conformance may only be totally guaranteed by formal proof, exhaustive testing with respect to some criteria or a combination of both.

Formal proof techniques are often very difficult to apply because most of them require a high level of mathematical skills. In order to make proof techniques available to non-specialists, significant efforts have been spent in the last years

⋆ This work has been partially funded by the RNTL POSE project, supported by the French Ministry of Research and New Technologies.
to provide automatic deductive proof tools. For instance, the JACK tool [2] has been designed to automatically prove the correctness of some Java code with respect to its JML specification. JML (Java Modeling Language [9]) is a formal specification language for Java, based on assertions such as invariants, pre- and post-conditions. During the proof process, JACK firstly produces the proof obligations required to show that the implementation (i.e. the code) conforms to its specification. Then, JACK tries to automatically prove each of them.

Nevertheless a difficulty remains; when a theorem is not automatically proved, there are two possibilities: either the theorem is correct and the prover is unable to decide it, or the theorem is just simply incorrect.

In these cases, testing may contribute to exhibiting that there is an error in the code (or in the specification), or increase the confidence that the theorem is correct (meaning that the implementation conforms to its specification). To achieve that, our proposal is to produce a huge number of tests, with combinatorial testing techniques. Then, the relevance of the test suite has to be evaluated. Indeed, if the test cases are not related to the part of the program involved in the proof obligation, they cannot provide feedback about the correctness of the theorem to be proved. In this work, the relevance of the test suite is evaluated with coverage measurements.

Section 2 introduces the principles of JML, an executable model-based specification language. Section 3 gives an overview of the JACK prover. A testing process is described in Sect. 4 along with a brief presentation of TOBIAS tool. Section 5 and 6 present the evaluation of the quality of the test suite by respectively using coverage and mutation techniques. Finally Section 8 draws the conclusions and presents the perspectives of this work.

2 Using JML as a Specification Language

2.1 A Small Example

We use a simple buffer system as a running example. This system is composed of three buffers. Each buffer is modeled by an integer value, which indicates the number of elements in it. The system state is given by the three variables, b1, b2 and b3. The maximum size of the system is 40 elements. The system has to distribute the elements among the buffers so that: buffer b1 is smaller or equal than b2, which is smaller or equal than b3. The difference between b1 and b3 should not exceed 15 elements. These constraints leave some freedom in the manner the elements can be shared between the buffers. For example, 30 elements can be stored with b1=5 b2=10 b3=15 or with b1=8 b2=10 b3=12.

Three methods are provided to modify the system. Init resets all buffers to zero. Add(x) increases the total number of elements of the system by x (x > 0) by adding x elements to the buffers; these elements are distributed in b1, b2, and b3. Remove(x) decreases the total number of elements in the system by x (x > 0) by removing x elements from the buffers.
2.2 The Java Modeling Language

The Java Modeling Language (JML) [10] is an annotation language used to specify Java programs by expressing formal properties and requirements on the classes and their methods. The Java syntax of JML makes it easier for Java programmers to read and write specifications. The core expression of the language is based on Java, with new keywords and logical constructions.

We illustrate the JML syntax on the buffer example given in Fig. 1. The JML specification appears within special Java comments, between /*@ and @*/ or starting with //@. The specification of each method precedes its interface declaration. This follows the usual convention of Java tools, such as JavaDoc, which put such descriptive information in front of the method.

JML annotations rely on three kinds of assertions: class invariants, preconditions and postconditions. Invariants have to hold in all visible states. A visible state roughly corresponds to the initial and final states of any method invocation [9]. The invariant stated in Fig. 1, line 6, indicates that the maximum size of the system is 40 elements. JML relies on the principles of Design by Contract [12] which states that to invoke a method, the system must satisfy the method precondition, and as a counterpart, the method has to establish its postconditions.

A method’s precondition is given by the requires clause. In our example, Init precondition is set to true (see Fig. 1, line 22). The preconditions of Add and Remove (lines 32 and 46) are such that the number of element to be added into (resp. removed of) the system is positive, less or equal to 5 and does not cause buffer overflow (resp. underflow).

Postconditions are expressed in the ensures clauses. For instance, the postconditions of Add and Remove (Fig. 1, lines 34 and 48) express that the add and remove operations are correctly performed.
JML extends the Java syntax with several keywords. `\result` denotes the return value of the method. It can only be used in `\ensure` clauses of a non-void method. `\old{Expr}` (Fig. 1, lines 34 and 48) refers to the value that the expression `Expr` had in the initial state of a method. `\forall` and `\exists` designate universal and existential quantifiers.

3 JACK Proof Obligation Generator

The Java Applet Correctness Kit (or JACK) [2] provides an environment for the verification of Java and JavaCard programs annotated with JML. JACK aims at proving properties of a given Java class, considered in isolation; these properties are expressed as JML assertions. It implements a fully automated weakest precondition calculus that generates proof obligations (POs) from annotated Java sources. Each proof obligation is related to a path in the source code of the program.

Those proof obligations can be discharged using different theorem provers. The Jack proof manager sends the proof obligations to the different provers, and keeps track of proved and unproved proof obligations. Currently proof obligations can be generated for the B method’s prover (developed by Clearsy), the Simplify theorem prover (notably used by ESC/Java), the Coq proof assistant, and PVS. For the case studies we present in this paper, Simplify was used.

JACK consists of two parts: (1) a converter, which is a lemma generator from Java source annotated with JML into adequate formalism (such as B lemmas, for B prover), and (2), a viewer, which allows developers to understand the generated lemmas. The mathematical complexity of the underlying concepts is hidden. JACK provides a dedicated proof obligation viewer that presents the proof obligations connected to execution paths within the program. Goals and hypotheses can be displayed in a Java/JML like notation.

Buffer example Figure 1 provides an incorrect implementation of the Buffer specification. Indeed, the `Remove` method is incorrect because the statement in line 57 may set buffer `b1` to a negative value while keeping the total number of elements positive, which is forbidden by the class invariant.

For this example, Simplify (version 1.2.0) was not able to prove 8 proof obligations. Some of them correspond to correct code and require the user to add further elements in the specification to help the prover. Others correspond to the erroneous implementation of `Remove`, and the prover will never be able to establish them. Table 1 illustrates each proof obligation and the corresponding specification and code lines. Figure 2 gives all proof obligations related to the invariant preservation by the `Remove` method.

4 The Testing Process

JML specifications can be used as oracle for a test process. In this section, we first cover some principles of conformance testing with JML, before introducing the TOBIAS tool.
### Table 1. Buffer specification/code lines related to improved PO

<table>
<thead>
<tr>
<th>Method</th>
<th>PO specification line</th>
<th>code lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>Add</td>
<td>3</td>
<td>34</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>9</td>
</tr>
<tr>
<td>Remove</td>
<td>6</td>
<td>48</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>7</td>
</tr>
</tbody>
</table>

#### 4.1 JML as a Test Oracle

JML has an executable character. It is possible to use invariant assertions, as well as pre- and postconditions as an oracle for conformance testing. JML specifications are translated into Java by the `jmlc` tool, added to the code of the specified program, and checked against it, during its execution.

The executable assertions are thus executed before, during and after the execution of a given operation. Invariants are properties that have to hold in all visible states. A visible state roughly corresponds to the initial and final states of any method invocation [9]. When an operation is executed, three cases may happen. **All checks succeed:** the behavior of the operation conforms with the specification for these input values and initial state. The test delivers a PASS verdict. **An intermediate or final check fails:** this reveals an inconsistency between the behavior of the operation and its specification. The implementation does not conform to the specification and the test delivers a FAIL verdict. **An initial check fails:** in this case, performing the whole test will not bring useful information because it is performed outside of the specified behavior. This test delivers an INCONCLUSIVE verdict. For example, if $\sqrt{x}$ has a precondition that requires $x$ to be positive. Therefore, a test of a square root method with a negative value leads to an INCONCLUSIVE verdict.

#### 4.2 Test Cases Definition

We define a test case as a sequence of operation calls. For example, in the following, test case TC1 initializes the buffer system, adds two elements and removes one of them.
Fig. 2. One proof obligation related to the implementation given Fig. 1

TC1 : Init() ; Add(2) ; Remove(1)
TC2 : Init() ; Add(-1)
TC3 : Init() ; Add(2) ; Remove(3)
TC4 : Init() ; Add(3) ; Remove(2) ; Remove(1)

Each operation call may lead to a PASS, FAIL or INCONCLUSIVE verdict. As soon as a FAIL or INCONCLUSIVE verdict happens, we choose to stop the test case execution and mark it with this verdict. A test case that is carried out completely receives a PASS verdict.

In the context of the buffer specification, the test cases TC2 and TC3 should produce an INCONCLUSIVE verdict: in TC2, Add is called with an incorrect negative parameter, in TC3, one tries to remove more than what has been added so far. If tests TC1 and TC4 are executed against a “correct” implementation, they should produce a PASS.

4.3 Test Case Generation

Combinatorial testing performs combinations of selected input parameters values for given operations and given states. For example, a tool like JML-JUnit [3] generates test cases which consist of a single call to a class constructor, fol-
owed by a single call to one of the methods. Each test case corresponds to a combination of parameters of the constructor and parameters of the method.

TOBIAS is a test generator based on combinatorial testing [4]. It adapts combinatorial testing to the generation of sequences of operation calls. The input of TOBIAS is composed of a test pattern (also called test schema) which defines a set of test cases. A schema is a bounded regular expression involving the Java methods and their associated JML specification. TOBIAS unfolds the schema into a set of sequences, then computes all combinations of the input parameters for all operations of the schema.

The schemas may be expressed in terms of groups, which are structuring facilities that associate a method, or a set of methods, to typical values. Groups may also involve several operations. Let \( S_2 \) be a schema:

\[
S_2 = \text{BufGr}^{1..3} \text{ with } \text{BufGr} = \{\text{Init}()\} \cup \{\text{Add}(x)|x \in \{1, 2, 3, 4, 5\}\} \cup \{\text{Remove}(y)|y \in \{2, 3, 5\}\}
\]

\text{BufGr} is a set of \((1+5+3)=9\) instantiations. The suffix \(^{1..3}\) means that the group is repeated 1 to 3 times. \( S_2 \) is unfolded into \( 9+(9*9)+(9*9*9)=819 \) test sequences.

Several case studies [1, 6] have shown that TOBIAS increases the productivity of the test engineer by allowing him to generate thousands of test cases from a few lines of schema description.

TOBIAS includes tools to turn such a test suite into a JUnit file. The execution of constructor Buffer is automatically added at the beginning of all test sequences. Executing this test suite against the erroneous buffer implementation (given in Fig. 1) reveals failures: 17 tests fail, 378 succeed and 424 are inconclusive. All failing tests report that the error occurs when checking line 7 of the invariant after the execution of Remove. This corresponds to PO #8. JML does not allow to point out a particular statement where the error was introduced because assertions are only checked at the exit of the operation.

5 Test Coverage Measurement and Proof Obligations

At this point of the study, we know that there is an error in operation Remove, and that PO #8 is false. Can we get more confidence from the tests that the remaining POs are correct?

Line coverage reports whether each executable statement is encountered. It is also known as statement coverage, segment coverage [14], C1 and basic block coverage. Basic block coverage is the same as statement coverage except that the unit of code measured is each sequence of non-branching statements.

For the 819 test cases generated from \( S_2 \) JCoverage [8] reports that 100% of the Java statements have been executed. So, at least all the operations have been covered, and all JML assertions have been evaluated while exiting these operations. But, at this point, nothing guarantees that the path of each proof obligation has been covered by a test.
To evaluate if a test case \( TC_i \) is relevant w.r.t. an unproved PO \( PO_k \), a first simple idea is to evaluate the line coverage associated with \( TC_i \) and to compare it with the lines of the \( PO_k \) path. As a first approximation, if \( TC_i \) does not execute all code lines associated to \( PO_k \), then it does not follow the same path, and \( TC_i \) is then not relevant to have an idea on \( PO_k \)'s correctness. If \( TC_i \) executes at least all code lines associated to \( PO_k \), then it may follow the same path. So \( TC_i \) may be relevant to increase the confidence in \( PO_k \)'s correctness.

For the Buffer example, we have executed the 819 test cases and we have analysed their line coverage with JCoverage. We gathered test cases with respect to their line code coverage into 25 “packets”, as described in Table 2. All test cases of a given packet cover the same lines (presumably with different values).

All 17 failed tests belong to packet #15. A closer look at the coverage of packets #14 to #17 shows that they share the same coverage of the Buffer constructor and operation Add. We can also notice that, among these 4 packets, line 57 is only executed in packet #15. We know that the error was related

<table>
<thead>
<tr>
<th>Packet</th>
<th>lines of code</th>
<th># of TC</th>
<th>PO</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>16-20, 27-30</td>
<td>3</td>
<td>1,2</td>
</tr>
<tr>
<td>2</td>
<td>16-20, 27-30, 37, 40, 43-44</td>
<td>70</td>
<td>1,2,3,4,5</td>
</tr>
<tr>
<td>3</td>
<td>16-20, 27-30, 37, 40, 43-44, 51, 54, 57-58</td>
<td>54</td>
<td>1,2,3,4,5,6,7,8</td>
</tr>
<tr>
<td>4</td>
<td>16-20, 27-30, 37, 40, 43-44, 51-52, 58</td>
<td>16</td>
<td>1,2,3,4,5</td>
</tr>
<tr>
<td>5</td>
<td>16-20, 27-30, 37, 40-41, 43-44</td>
<td>30</td>
<td>1,2,3,4,5</td>
</tr>
<tr>
<td>6</td>
<td>16-20, 27-30, 37-38, 44, 41, 51, 54, 57-58</td>
<td>20</td>
<td>1,2,6,7,8</td>
</tr>
<tr>
<td>7</td>
<td>16-20, 27-30, 51, 54, 57-58</td>
<td>30</td>
<td>1,2,6,7,8</td>
</tr>
<tr>
<td>8</td>
<td>16-20, 27-30, 51, 54-55, 57-58</td>
<td>12</td>
<td>1,2,6,7,8</td>
</tr>
<tr>
<td>9</td>
<td>16-20, 37, 40, 43-44</td>
<td>8</td>
<td>1,2</td>
</tr>
<tr>
<td>10</td>
<td>16-20, 37, 40, 43-44, 51, 54, 57-58</td>
<td>32</td>
<td>1,2,3,4,5,6,7,8</td>
</tr>
<tr>
<td>11</td>
<td>16-20, 37, 40, 43-44, 51, 54-55, 57-58</td>
<td>23</td>
<td>1,2,3,4,5,6,7,8</td>
</tr>
<tr>
<td>12</td>
<td>16-20, 37, 40-41, 43-44, 51-52, 54, 57-58</td>
<td>38</td>
<td>1,2,3,4,5,6,7,8</td>
</tr>
<tr>
<td>13</td>
<td>16-20, 37, 40, 43-44, 51-52, 58</td>
<td>71</td>
<td>1,2,3,4,5</td>
</tr>
<tr>
<td>14</td>
<td>16-20, 37, 40-41, 43-44</td>
<td>102</td>
<td>1,2,3,4,5</td>
</tr>
<tr>
<td>15</td>
<td>16-20, 37, 40-41, 43-44, 51, 54, 57-58</td>
<td>31</td>
<td>1,2,3,4,5,6,7,8</td>
</tr>
<tr>
<td>16</td>
<td>16-20, 37, 40-41, 43-44, 51, 54-55, 57-58</td>
<td>14</td>
<td>1,2,3,4,5,6,7,8</td>
</tr>
<tr>
<td>17</td>
<td>16-20, 37, 40-41, 43-44, 51-52, 58</td>
<td>18</td>
<td>1,2,3,4,5</td>
</tr>
<tr>
<td>18</td>
<td>16-20, 37-38, 40, 43-44, 51, 54, 57-58</td>
<td>76</td>
<td>1,2,3,4,5,6,7,8</td>
</tr>
<tr>
<td>19</td>
<td>16-20, 37, 40-41, 44, 51, 54-55, 57-58</td>
<td>12</td>
<td>1,2,6,7,8</td>
</tr>
<tr>
<td>20</td>
<td>16-20, 37-38, 40-41, 43-44</td>
<td>35</td>
<td>1,2,3,4,5</td>
</tr>
<tr>
<td>21</td>
<td>16-20, 37-38, 44, 51, 54, 57-58</td>
<td>63</td>
<td>1,2,6,7,8</td>
</tr>
<tr>
<td>22</td>
<td>16-20, 37-38, 44, 51, 54-55, 57-58</td>
<td>12</td>
<td>1,2,6,7,8</td>
</tr>
<tr>
<td>23</td>
<td>16-20, 51, 54, 57-58</td>
<td>6</td>
<td>1,2,6,7,8</td>
</tr>
<tr>
<td>24</td>
<td>16-20, 51, 54-55, 57-58</td>
<td>23</td>
<td>1,2,6,7,8</td>
</tr>
<tr>
<td>25</td>
<td>16-20, 51-52, 54-55, 57-58</td>
<td>10</td>
<td>1,2,6,7,8</td>
</tr>
</tbody>
</table>

| total of TC | 819 |

**Table 2.** 25 packets for the 819 Buffer example tests
to operation \textbf{Remove}; this closer look at coverage information suggests that the error is located at line 57.

Still, our main concern is to increase our confidence in unproved POs. Table 2 tells us that all test cases were related to POs #1 and #2. More than 500 test cases are related to POs #3, #4 and #5, and more than 400 test cases are related to POs #6, #7 and #8. Since failed tests were only related to PO #8, we can increase our confidence in the unproved POs, since each of them has been tested several hundred times\footnote{To be more accurate, we should have excluded inconclusive tests from Table 2. We intend to include this improvement in a future version of our reporting tool.} and did not reveal any error.

6 Test Relevance Analysis using Mutation

Another way to measure the quality of the test set is to evaluate its fault detection capabilities. Mutation analysis can be used for this purpose.

Mutation analysis is based on seeding the implementation with a fault by applying a mutation operator, and checking whether test set identifies this fault or not [5, 11]. A mutated program is called a mutant. A mutant is said to be killed if the test suite reveals its error.

An appropriate set of mutation operators should be representative of classical programming errors. The idea behind mutation testing is quite simple: if a test suite kills all mutants generated by these operators then, since it is able to find these small differences, it is likely to be good at finding real faults.

When using mutation programs such as MuJava [11, 13], two kinds of problems may arise. First, applying a large set of mutation operators to a real-size program usually results in a huge number of mutants. Secondly, some of the mutants are actually equivalent to the original program and can not be killed.

In order to limit the number of mutants, we applied mutations only to statements that are involved in the path related to unproved POs. For instance, we generated 20 mutants corresponding to the unproved PO #7 and our test suite killed 100\% of them. An interesting point is that different tests of a same packet may kill different mutants. This means that these packets feature some kind of diversity.

At this point of the case study, we have reached sufficient confidence in the correctness of the remaining proof obligations to get back to an interactive proof activity. Actually, only POs #1 to #5 deserve to be proved at this stage, because corrections of \textbf{Remove} will not affect their correctness. Of course, nothing guarantees that our test suite was able to detect all kinds of subtle errors. This is why a final proof activity is definitely needed to assess program correctness. Still, the benefit of our testing activity is that the validation engineer will not waste time trying to prove false proof obligations, or even correct ones such as #6 or #7 which may be affected by the correction of \textbf{Remove}.
7  Banking Application Case Study

*Industrial Case Study.* The combination of the proof/test processes was experimented on an industrial case study provided by Gemplus (a smart card manufacturer). The case study is a banking application which deals with money transfers [1]. It has been produced by Gemplus Research Labs, and used for the first experimentation of JACK. This case study is somehow representative of Java applications connected to smart cards. The application user (i.e. the customer) can consult his accounts and make some money transfers from one account to another. The user can also record some “transfer rules”, in order to schedule regular transfers. These rules can be either saving or spending rules.

The case study is actually a simplified version of an application already used in the real world, written with 500 LOC, distributed into 8 classes. The specification is given in JML. Most preconditions are set to true. Since the application deals with money, and since some users may have malicious behaviors, the application is expected to have defensive mechanisms. Thus, it is supposed to accept any entry, but it should return error messages or raise exceptions if the inputs are not those expected for a nominal behavior.

In order to evaluate our approach, we worked on a correct version of the program, and introduced an error in the `Currency_src` class.

*Proof Process.* We used JACK with Simplify for this example. For some unknown reason, we were only able to compile 7 of the 8 classes with the JACK tool. Table 3 reports on the total number of POs generated by JACK and the number of POs which remained unproved after the automatic proof process.

*Test Process.* For each class, we produced only one TOBIAS schema. They were rather straightforward, as $S2$ in the previous case study. Their design and unfolding with the TOBIAS tool only took us a few minutes. Each schema produced between 48 and 1024 test cases. We then executed them, and, as expected, failed tests were only related to `Currency_src`.

*Test Coverage and Proof Obligations.* We grouped our tests into packets on the basis of statement coverage. As Table 3 reports, most classes correspond to a small number of packets. This motivates further research using some path coverage tool instead of statement coverage, in order to have a more accurate distribution of test cases.

*Killing Mutants.* The mutation analysis was not possible for `Account` and `Rule` due to unsolved technical problems. For the other classes, we could notice that all mutants were killed for `Balances_src` and `Transfers_src`. However, no mutant have been killed for `SavingRule` and `SpendingRule`. Clearly, testing schemas for those two classes were not relevant enough. More insightful test schemas must be defined to generate appropriate test suites for these classes and increase the confidence in their correctness.
<table>
<thead>
<tr>
<th>Class</th>
<th># PO</th>
<th># unproved PO</th>
<th># TC</th>
<th># TC packets</th>
<th># Mutant</th>
<th># killed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Account</td>
<td>8</td>
<td>2</td>
<td>84</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Balances_src</td>
<td>114</td>
<td>96</td>
<td>72</td>
<td>4</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>Currency_src</td>
<td>92</td>
<td>16</td>
<td>244</td>
<td>32</td>
<td>17</td>
<td>12</td>
</tr>
<tr>
<td>Rule</td>
<td>17</td>
<td>9</td>
<td>72</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SavingRule</td>
<td>62</td>
<td>31</td>
<td>48</td>
<td>2</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>SpendingRule</td>
<td>37</td>
<td>20</td>
<td>48</td>
<td>2</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>Transfers_src</td>
<td>1170</td>
<td>560</td>
<td>1024</td>
<td>3</td>
<td>81</td>
<td>81</td>
</tr>
<tr>
<td>AccountMan_src</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3. Banking example

8 Conclusion and Perspectives

This paper has proposed an approach to combine a proof process with testing activities. The goal is to help the validation engineer decide on the correctness of unproved proof obligations. Our testing process starts from a combinatorial testing phase where hundreds of test cases are generated with small design efforts (few schemas can generate several thousand of test cases). Their execution may reveal errors in the code under validation and hence point out false proof obligations. The huge number of succeeded tests, and an evaluation of the quality of the test suite, should increase the confidence in the remaining proof obligations.

Two techniques are proposed to evaluate the quality of the test suite: a comparison of statement coverage with the paths related to unproved proof obligations, and an assessment of the fault capabilities based on mutation testing. The first evaluation technique results in a distribution of the test suite into “packets” of tests which cover the same set of statements. The second evaluation restricts mutations to those which hit statements appearing in the path of unproved PO. The assessment of the quality of the test suite can be further refined by crossing information gathered from these two techniques. Different tests grouped in the same packet exhibit more “diversity” if they kill different mutants.

The approach was experimented on two case studies in the context of Java/JML. We used the JACK proof environment, the TOBIAS test generator, and JCov erage and MuJava for quality assessment.

We divide the future work into four points as below:

Statement vs Path coverage. Since JACK is based on the notion of path, it makes sense to use path coverage instead of statement coverage. Besides the fact that we do not have such a tool available in our environment, we suspect that this more detailed analysis will slow down the testing process, and may in some cases result into over-detailed test reports. Therefore, we believe that it should be provided as an option.

Automatic process. Our approach only makes sense if the whole testing process is cheaper than interactive proof activities. Here each step is automated. JACK associates automatically PO to paths. Tests can be generated automatically thanks
to combinatorial tools, such as TOBIAS. JCoverage analyses automatically lines covered during test execution. Grouping test cases is done by sorting JCoverage results. Killing mutants is done automatically.

Reporting. In the previous section, we mentioned the potential interest of crossing packet information with mutation scores. The tool which will compute this information has still to be developed. Further effort should be dedicated to provide the user with synthetic reports on the quality of his tests (describing how relevant tests are with respect to coverage, mutation analysis, ...).

Feeding assertions into the proof process. The tests generated with TOBIAS are designed independently of the structure of the code or the specification. We expect that they could provide interesting input to the Daikon invariant generator [7]. This would allow to feedback of the proof process with assertions generated from the tests, resulting in a secondary benefit of the testing activity.

References


77
Automated Verification of Completeness and Consistency of Abstract State Machine Specifications using a SAT Solver

Martin Ouimet and Kristina Lundqvist

Embedded Systems Laboratory
Massachusetts Institute of Technology
Cambridge, MA, 02139, USA
{mouimet, kristina}@mit.edu

Abstract. In the requirements engineering community, consistency and completeness have been identified as important properties of system specifications. Custom algorithms to verify these properties automatically have been devised for a number of specification languages, including SCR, RSML, and Statecharts. In this paper, we provide means to automatically verify completeness and consistency of Abstract State Machine (ASM) specifications. The verification is performed using a widely available tool, a SAT solver. The use of a SAT solver removes the need for designing and fine tuning language specific verification algorithms. Furthermore, the use of a SAT solver automates the verification procedure and produces a counterexample automatically when a specification is incomplete or inconsistent. We provide an algorithm to translate ASM specifications to a SAT problem instance. The translation is illustrated using the TASM toolset in conjunction with the “production cell system” case study.

1 Introduction

Consistency and completeness were identified as useful properties of specifications in [1] and in [2]. In the context of specification of embedded systems, completeness of the specification is defined as the specification having a response for every possible class of inputs. In the same context, consistency is defined as the specification being free of contradictory behavior, including unintentional non-determinism [2]. Formal definitions of these properties, in the context of Abstract State Machine (ASM) specifications, are given in Section 3. Traditionally, verifying these properties was accomplished manually by system specifiers, through inspection of specifications. Because a specification is likely to evolve during the engineering lifecycle, the ability to verify these properties automatically can ease and shorten the analysis process [2]. Language specific verification algorithms have been proposed in [1] and in [2]. In contrast, the approach proposed in this paper is not language specific and can be reused for other languages. The proposed approach achieves verification by translating specifications to formulas in propositional logic, formulating completeness and consistency as a boolean satisfiability problem (SAT) [3], and automating the verification procedure by using a generally available solver.
Abstract State Machines (ASM) have been used to specify, analyze, and verify hardware and software systems at different levels of abstraction [4]. Abstract State Machines have also been used to automate engineering activities, including verification using model checkers [5] and test case generation [6]. The Timed Abstract State Machine (TASM) language is an extension of the ASM language that includes facilities for specifying non-functional properties, namely time and resource consumption [7]. The TASM language and its associated toolset have been used to model and simulate real-time systems [8], [9], and [10]. The relationship between ASM and TASM is quite close and the notions of completeness and consistency introduced in this paper are equally applicable to both ASM and TASM.

In this paper, an approach to automatically verify consistency and completeness of TASM specifications is provided. The verification is achieved by mapping TASM specifications to boolean formulas in Conjunctive Normal Form (CNF). The specified mapping is derived using the structural properties of the specification and does not require the generation of a global reachability graph, thereby avoiding the infamous state space explosion problem [2]. The proposed mapping could also be applied to ASM specifications because the mapping does not consider the extensions of the TASM language. The mapping to boolean formulas in CNF allows for automated verification using a $SAT$ solver. The mapping is achieved in such a way that consistency and completeness are expressed as unsatisfiability of the boolean formulas. If the TASM specification is incomplete or inconsistent, the $SAT$ solver will generate an assignment which makes the boolean formula satisfiable. This assignment serves as the counterexample to exercise the incompleteness or inconsistency of the specification.

2 Related Work

The definition and automated verification of completeness and consistency of specifications were introduced in [1] and in [2]. In [1], the RSML language, a hierarchical state-based language, is used to express requirements. The language is automatically analyzed for completeness and consistency using an algorithm specifically targeted for the RSML language. In [2], a similar approach is used for analysis of requirements expressed in the SCR language. These two approaches rely on specific purpose algorithms for the efficient and automated analysis of consistency and completeness. Consequently, the proposed algorithms cannot be reused for other languages. In contrast, the approach proposed in this work utilizes a general purpose solver, a $SAT$ solver. The proposed translation from TASM specifications to boolean formulas in CNF can be reused for other specification languages. The use of a $SAT$ solver guarantees that the analysis procedure is optimized.

In the ASM community, various derivatives of the ASM language have been developed, including the ASM Workbench [11] and the Abstract State Machine Language (AsmL) [12]. A mapping between the ASM Workbench language (ASM-SL) and finite state machines, for the purpose of model checking, was proposed in [5]. A mapping between the AsmL language and finite state machines was proposed in [6]. The mapping to finite state machines was used for automated test case generation [13]. The mapping proposed in this paper resembles the mappings proposed in these two approaches ex-
cept that it ignores the effect of rule applications and does not need to generate a global reachability graph. The proposed mapping concerns itself only with relationships between rule guards inside a single machine and hence produces a smaller state space than might be generated through a complete reachability graph.

SAT solvers have been used for a variety of automated analysis, including test case generation [14], [15]. Although the SAT problem is known to be NP-Complete, the use of SAT solvers has been shown to be useful in a wide range of cases. SAT solvers and model checkers show similarities in their benefits, namely automation of the verification procedure and automation of the counterexample generation. SAT solvers and model checkers also show similarities in their drawbacks, namely the potential for state space explosion and the resulting intractability of large state space exploration.

### 3 Definitions

Abstract State Machines (ASM) is a formal language and an associated methodology for system design and analysis. The language and methodology have been used to model a variety of hardware and software systems, at different levels of abstraction [16]. There have been a number of derivatives of the ASM language, including the ASM Workbench language (ASM-SL) [11], the Abstract State Machine Language (AsmL) [12], and the Timed Abstract State Machine (TASM) language [7]. While these derivatives have syntactic and semantic differences, they all rely on the basic concepts of ASMs. The abstract state machine formalism revolves around the concepts of an abstract machine and an abstract state. System behavior is specified as the computing steps of the abstract machine. A computing step is the atomic unit of computation, defined as a set of parallel updates made to global state. A state is defined as the values of all variables at a specific step. A machine executes a step by yielding a set of state updates. A run, potentially infinite, is a sequence of steps. The structure of an ASM is a finite set of rules, written in precondition-effect style. For an ASM that contains \( n \) rules, a block machine, also called a machine in canonical form has the following structure:

\[
R_1 \equiv \text{if } G_1 \text{ then } E_1 \\
R_2 \equiv \text{if } G_2 \text{ then } E_2 \\
\vdots \\
R_n \equiv \text{if } G_n \text{ then } E_n
\]

The guard \( G_i \) is the condition that needs to be enabled for the effect of the rule, \( E_i \), to be applied to the environment. The effect of the rule is grouped into an update set, which is applied atomically to the environment at each computation step of the machine. For a complete description of the theory of abstract state machines, the reader is referred to [16].

#### 3.1 The Timed Abstract State Machine (TASM) Language

The Timed Abstract State Machine (TASM) language [7] is an extension of the ASM language for the specification and analysis of real-time systems. The TASM language
extends the specification of rules by enabling the specification non-functional properties, namely time and resource consumption. The semantics of rule execution extend the update set concept by including the duration of the rule execution and a set of resource consumptions during the rule execution.

In the TASM language, the canonical form given in equation 1 remains the same, except for the effect expressions. In the TASM language, the effect expressions, $E_i$, are extended to reflect the time and resource consumption specification. Other features of the TASM language include hierarchical and parallel composition, expressed through the use of sub machines and function machines. The definition of consistency and completeness, in terms of TASM specifications, is expressed as relational properties between the rule guards $G_i$. Consequently, the definitions of consistency and completeness given in Section 3.3 and Section 3.4, as well as the verification approach, are equally applicable to the TASM language and other derivatives of the ASM language. The approach could also be applicable to other languages as well, as long as these languages can be expressed in the canonical form given in equation 1. For a complete description of the TASM language, the reader is referred to [17].

### 3.2 The Satisfiability Problem

The satisfiability problem, also known as SAT for short, is the archetypical NP-Complete problem in the theory of computation [3]. The problem involves determining whether a boolean formula is satisfiable. A boolean formula is composed of a set of atomic propositions and operations. Atomic propositions are boolean variables that can take the values TRUE or FALSE. The propositions are connected using parentheses and the operations NOT, AND, and OR, represented by the symbols $\neg$, $\land$, and $\lor$. A boolean formula is satisfiable if there is an assignment of values to propositions which makes the formula TRUE. If no such assignment exists, the formula is unsatisfiable. A sample SAT problem is shown below:

$$(b_1 \lor b_2) \land (b_1 \lor b_3)$$

### 3.3 Completeness

Informally, completeness is defined as the specification having a response for every possible input combination. In the TASM world, for a given machine, this criteria means that a rule will be enabled for every possible combination of its monitored variables. The monitored variables are the variables in the environment which affect the machine execution. Formally, the disjunction of the rule guards of a given machine must form a tautology. The letter $S$ is used to denote an instance of the SAT problem. The completeness problem can be expressed as a SAT problem in the following way:

For $n$ rules:

$$S \equiv \neg (G_1 \lor G_2 \lor \ldots \lor G_n)$$
The completeness problem is casted as the negation of the disjunction so that counterexamples can be generated by the SAT solver. If $S$ is satisfiable, all the assignments that make $S$ satisfiable can be automatically generated by the SAT solver. If $S$ is not satisfiable, the specification is complete.

### 3.4 Consistency

Informally, for a state-based specification, consistency is defined as no state having more than one transition enabled at the same time [1]. The definition given in [2] is similar but extended to include other properties of the specification such as syntactical correctness and type checking. The definition of consistency adopted in this approach is the same as in [1]. In terms of TASM specifications, this definition states that no two rules can be enabled at the same time. This definition will lead to a set of SAT problems to define consistency:

For each pair of rules $R_i, R_j$ where $1 \leq i < j \leq n$:

$$S \equiv G_i \land G_j$$

$$\text{ASM} = \begin{cases} 
\text{consistent} & \text{if } S \text{ not satisfiable} \\
\text{inconsistent} & \text{if } S \text{ satisfiable}
\end{cases}$$

This definition yields a set of $\binom{n}{2}$ SAT problems. The individual SAT problems can also be composed into a single SAT problem. As for completeness, the SAT problem is defined in such a way that if the specification is not consistent, a counterexample is automatically generated. If $S$ is satisfiable, all the assignments that make $S$ satisfiable can be automatically generated by the SAT solver.

### 4 Translation to SAT

The TASM language is a typed language that includes integer datatypes, boolean datatypes, and user-defined types. User-defined types are analogous to enumeration types in programming languages. The TASM language is a subset of the ASM language and does not include all of the constructs of the ASM language. For example, the choose construct is not part of TASM. The concepts from the ASM language included in the TASM language are the same as defined in [5]. The translation from TASM to SAT involves mapping the rule guards, $G_i$, to boolean propositions, $b_i$, in Conjunctive Normal Form (CNF). The following subsections explain how this translation is performed.
4.1 Boolean and User-Defined Datatypes

In the TASM language, user-defined datatypes and boolean datatypes are simple types that can take values for a finite set. Boolean variables can take one of two values (true or false). User-defined types can take one of multiple values, as defined by the user. In typical specifications, user-defined types rarely exceed five or six members.

The only operations defined for boolean and user-defined datatypes are the comparison operators, $=$ and $!=$. No other operator is allowed for boolean and user-defined datatypes. In the translation to SAT, we take the equality operator ($=$) to mean a non-negated proposition (e.g., $b_1$). The operator $!=$ is translated to mean a negated proposition (e.g., $\neg b_1$). The translation to SAT for these datatypes involves 2 steps. The first step is generating the at least one clause and the at most one clause for each variable of type boolean or of type user-defined type. The second step involves formulating the property to be verified as a clause in CNF, $S$, according to the definitions in Section 3.

The at least one clause ensures that the variable must take at least one value from its finite set. This clause is simply the disjunction of equality propositions for each possible value that the variable can take.

The at most one clause ensures that no two $b_i$'s can be true at the same time.

To illustrate the generation of the at least one and at most one clauses, the following type is introduced: $type1 := \{val_1, \text{val}_2, \ldots, \text{val}_n\}$. A variable of type boolean can be viewed as a variable of type $type1$ where $n = 2$. First, the set of propositions is generated. In SAT, a proposition is a single letter with a subscript (e.g., $b_i$). For a variable named $var$ of type $type1$, the following propositions would be generated, where the $b_i$'s represent the SAT atomic propositions and the right hand side represents the meaning of the proposition in the TASM context:

\[
\begin{align*}
    b_1 : var &= val_1 \\
    b_2 : var &= val_2 \\
    \vdots \\
    b_n : var &= val_n
\end{align*}
\]

The at least one clause, denoted $C_1$ for this variable would be:

\[
C_1 \equiv b_1 \lor b_2 \ldots \lor b_n
\]

The at least one clause ensures that at least one of the $b_i$'s must be true for the clause to be true. The at most one clause ensures that no two $b_i$'s can be true at the same time. The at most one clause, denoted $C_2$ is the conjunction of multiple clauses:

\[
C_2 \equiv (\neg b_1 \lor \neg b_2 \ldots \lor \neg b_n) \land
    (b_1 \lor \neg b_2 \ldots \lor \neg b_n) \land
    (\neg b_1 \lor b_2 \ldots \lor \neg b_n) \land
    \vdots \land
    (\neg b_1 \lor \neg b_2 \ldots \lor b_n)
\]
The at most one clause generates $n + 1$ clauses, one for the full negations of the propositions and one for each $n - 1$ negations of propositions. This combination ensures that at most one of the clauses can be true. The conjunction $C_1 \land C_2$, which is already in conjunctive normal form, serves to enforce the "exactly one value per variable" constraint, also called type enforcement. The rule guards are made up of propositions that already exist in the proposition catalog. For each rule guard in the problem formulation $S$, for each constraint in the guards, if the constraint is of the form $\text{var} = \text{val}$, its corresponding proposition $b_i$ is looked up in the catalog and substituted in the problem formulation $S$. If, on the other hand, the constraint is of the form $\text{var} \neq \text{val}$, the $b_i$ corresponding to $\text{var} = \text{val}$ is looked up in the proposition table and the constraint in the guard is substituted by its negation, $\neg b_i$. Once the substitution is done in the rule guards, the formulated problem $S$ is then converted to conjunctive normal form using the well-known algorithm in [3]. The result of this substitution and conversion to CNF yields $S$ with only atomic boolean propositions. The full SAT problem can then be formed by the conjunction of $S$, $C_1$, and $C_2$:

$$\text{Full SAT problem} \equiv S \land C_1 \land C_2$$

### 4.2 Integer Datatypes

Similarly to boolean datatypes and user-defined datatypes, integer datatypes take values from a finite set. However, the number of values that integers can take is much larger than for boolean datatypes and much larger than for typical user-defined types. For example, in the TASM language, integers range from -32,768 to 32,767. While the approach suggested above for boolean and user-defined types might also work for integer types, the enumeration of all 65,536 possible values would be intractable for a single integer variable. The adopted mapping for integer variables relies on the fact that even though integers are used in TASM specifications, they are used in such a way that they could be replaced by user-defined types. In other words, in TASM specifications, the full range of integers is typically not used.

Nevertheless, integer datatypes are more complex than boolean and user-defined types because more operations are defined for integer datatypes. These operations are comparison operators and arithmetic operators. The comparison operators are $=, \neq, <, \leq, >, \text{and} \geq$. The arithmetic operators are $+, -, \ast, \text{and} /$. For the suggested translation, constraints on integer variables must be of the form $<\text{var}> <\text{comp\_op}> <\text{expr}>$, where $<\text{var}>$ is an integer variable $<\text{comp\_op}>$ is a comparison operator and $<\text{expr}>$ is an arbitrary arithmetic expression that can contain constants, variable references, function machine calls, and operators. The restriction is that the left hand side of constraints can contain only a variable, with no arithmetic expressions allowed. The translation proposed in this section, deals only with linear constraints whose right hand sides are constants. Arbitrary symbolic right hand sides will be handled in future work, as explained in section 6.

The key idea behind the translation is to convert each integer variable to a user-defined type. This is achieved by collecting all of the constraints on a given integer variable and extracting the intervals that are of interest. These intervals become the
members of the user-defined types. Once the integer type has been converted to a user-defined type in this fashion, it can then be converted to a boolean formula using the approach from Section 4.1. The algorithm to reduce integer variable to user-defined types consists of 4 steps. For each monitored variable of type integer:

1. Collect all constraints on the variable from $S$
2. Sort all constraints in ascending order of right-hand sides
3. Create unique intervals for constraints that overlap
4. In $S$, replace original constraints by disjunction of constraints for modified constraints in overlapping intervals

Once the integer variables have been reduced to user-defined types and the constraints in the problem formulation $S$ have been replaced with the appropriate combination of propositions, the full SAT instance can be created using the $at\ most\ one$ and the $at\ least\ one$ clauses, in the same fashion as explained in Section 4.1. For a specifications where there is significant use of integer constraints, the use of Mixed Integer Programming (MIP) solvers could be better suited for completeness and consistency analysis. This option is investigated in Section 6.

### 4.3 Complete Translation Algorithm

The basic translation principles have been explained in the previous sections. The complete translation algorithm can now be given, for a single machine:

1. Create problem instance $S$ depending on the property to be checked (consistency or completeness), as explained in Section 3
2. Replace function machine calls with extra rules
3. Replace symbolic right-hand sides with values from the chosen configuration
4. Reduce integer variables to user-defined type variables, as explained in Section 4.2
5. Iterate through all monitored variables and create $at\ least\ one$ clauses and $at\ most\ one$ clauses, as explained in section 4.1
6. Convert problem formulation $S$ to conjunctive normal form and create the full SAT instance, as explained in Section 4.1

### 5 Example

The translation presented in this paper is implemented in the TASM toolset. The resulting SAT problem is automatically analyzed using the open source SAT4J SAT solver [18]. The SAT4J solver is a Java-based solver which can be integrated seamlessly into any Java application. The TASM toolset provides the option to solve the completeness and consistency problems directly, without requiring the user to know that the specification is being translated to SAT. The toolset also provides the capability to "export" the generated SAT problem, so that the problem can be analyzed and solved outside of the toolset. The toolset was used to analyze the consistency and completeness of two examples, the production cell system [10] and an electronic throttle controller [8]. For these two examples, the performance of the translation algorithm...
and the feasibility of using a $SAT$ solver proved adequate. As an example, the translation for a machine definition is given. The sample machine specification is extracted from production cell system case study. The machine definition is for the 'loader' component, which is the component of the system responsible for putting blocks on the feed belt. The machine specification, expressed in the TASM language, is shown in Listing 1.

**Listing 1** Definition of the loader machine

```plaintext
R1: Empty Belt
{ t := 2;
  power := 200;
  if loaded_blocks < number - 1 and feed_belt = empty and feed_block = notavailable then
    feed_belt := loaded;
    loaded_blocks := loaded_blocks + 1;
    loader_sensor!;
}

R2: Load Last Block
{ t := 2;
  power := 200;
  if loaded_blocks = number - 1 and feed_belt = empty and feed_block = notavailable then
    feed_belt := loaded;
    loaded_blocks := loaded_blocks + 1;
    loader := notavailable;
    loader_sensor!;
}

R3: Loaded Belt
{ t := next;
  if feed_belt = loaded and loaded_blocks < number then
    skip;
}
```

For the verification of completeness, the translation to $SAT$, for initial conditions where $number = 5$, yielded 7 unique propositions:

\[
\begin{align*}
  b_1 &: \text{loaded\_blocks} \leq 3 \\
  b_2 &: \text{loaded\_blocks} = 4 \\
  b_3 &: \text{loaded\_blocks} \geq 5 \\
  b_4 &: \text{feed\_belt} = \text{empty} \\
  b_5 &: \text{feed\_belt} = \text{loaded} \\
  b_6 &: \text{feed\_block} = \text{available} \\
  b_7 &: \text{feed\_block} = \text{notavailable}
\end{align*}
\]

Once the mapping between TASM variable values and $SAT$ boolean propositions has been established, the rule guards, $G_i$, can be expressed in terms of boolean propositions. The completeness problem, $S$, is then constructed according to the definition is Section 3.3:
\[ G_1 \equiv b_1 \land b_4 \land b_7 \]
\[ G_2 \equiv b_2 \land b_4 \land b_7 \]
\[ G_3 \equiv b_5 \land (b_1 \lor b_2) \]
\[ S \equiv \neg(G_1 \lor G_2 \lor G_3) \]

The complete translation to \( SAT \), in CNF, yielded 13 total propositions:

\[
S \text{ in } CNF = \left\{ \begin{align*}
(\neg b_7 \lor \neg b_4 \lor \neg b_1) & \land \\
(\neg b_7 \lor \neg b_4 \lor \neg b_2) & \land \\
(\neg b_1 \lor \neg b_5) & \land \\
(\neg b_2 \lor \neg b_5) & \land \\
(b_1 \lor b_2 \lor b_3) & \land \\
(b_4 \lor b_5) & \land \\
(b_6 \lor b_7) & \land \\
(-b_1 \lor \neg b_2 \lor \neg b_3) & \land \\
(b_1 \lor \neg b_2 \lor \neg b_3) & \land \\
(-b_1 \lor b_2 \lor \neg b_3) & \land \\
(-b_1 \lor \neg b_2 \lor b_3) & \land \\
(-b_4 \lor \neg b_5) & \land \\
(-b_6 \lor \neg b_7) & \land
\end{align*} \right. 
\]

At least one clauses
\[
(\neg b_7 \lor \neg b_4 \lor \neg b_1) \land
(\neg b_7 \lor \neg b_4 \lor \neg b_2) \land
(\neg b_1 \lor \neg b_5) \land
(\neg b_2 \lor \neg b_5) \land
(b_1 \lor b_2 \lor b_3) \land
(b_4 \lor b_5) \land
(b_6 \lor b_7) \land
(-b_1 \lor \neg b_2 \lor \neg b_3) \land
(-b_1 \lor \neg b_2 \lor b_3) \land
(-b_4 \lor \neg b_5) \land
(-b_6 \lor \neg b_7)
\]

At most one clauses

The \( SAT \) problem resulting from the translation is relatively small and running it through the SAT4J solver yields a solution in negligible time. For this machine, the rule set is not complete. The TASM toolset uses the SAT4J solver to generate the counterexamples in which no rule is enabled. An assignment to propositions that makes the problem satisfiable is \((b_2 = \text{true}, \ b_4 = \text{true}, \ b_6 = \text{true})\) and all other propositions are assigned \textit{false}. In terms of the TASM specification, the counterexample which is generated is the set \((\text{loaded_blocks} = 4, \ \text{feed_belt} = \text{empty}, \ \text{feed_block} = \text{available})\). To check the consistency of the rule set for the 'loader' machine, the same set of proposition was generated, but the set of clauses grew to 159. However, many of the clauses were redundant, due to the long form used for the conversion to CNF. Future work in tool development will improve the translation to CNF by removing redundant clauses. Nevertheless, the \( SAT \) problem was verified to be unsatisfiable in negligible time. In other words, the rules of machine 'loader' are consistent. The preliminary results from the translation algorithm indicate that the performance of the translation algorithm might overshadow the performance of the \( SAT \) solver.

6 Conclusion and Future Work

In this paper, a translation from Abstract State Machine (ASM) specifications to the boolean satisfiability problem is given. The translation is performed in the context of
the Timed Abstract State Machine (TASM) language, but the translation is equally applicable to standard ASM specifications and ASM derivatives. The translation is used to investigate completeness and consistency of the specification, for a single abstract state machine. Completeness and consistency of specifications were identified as important properties of specifications. The ability to verify these properties automatically, using a widely available and optimized tool, a SAT solver, is provided. This approach contrasts previous attempts using other languages, which have used special purpose verification algorithms. Previous attempts have motivated the use of special purpose algorithms to remove the need to generate a global reachability graph, as would be done in approaches based on model checkers. The translation proposed in this work also removes the need to generate a global reachability graph by constraining the analysis to a single machine and by considering only the structural properties of the specification. The big open question in this work is whether the use of a SAT solver to verify consistency and completeness is feasible for archetypical real-time system specifications. The number of propositions can grow exponentially, depending on the nature of the specification. Preliminary results indicate that the translation algorithm could undergo further optimization since it appears to be a bottleneck, compared to the time spent in the SAT solver. The translation algorithm will be analyzed in detail for time complexity and will be optimized accordingly.

6.1 Future Work

The translation given in this work maps TASM specifications to boolean formulas. The use of boolean formulas makes negation and manipulation of rule guards straightforward. The translation will be used for model-based test case generation, using existing approaches [14] [15] and existing coverage criteria for ASM specifications [19]. For rule guards that contain multiple integer variables, the use of SAT solvers might not be the most optimal approach. Mixed Integer Programming (MIP) solvers such as the open source GNU Linear Programming Kit (GLPK) could provide a viable alternative. The translation to an MIP solver would not require the reduction of the integer variables as proposed in this work since MIP solvers can handle a mix of boolean variables and integer variables. However, using an MIP solver would require a reformulation of the problem because the input of such solvers requires a conjunction of constraints. Handling disjunction of constraints can be expressed, using modeling tricks such as the “big M” approach [20] and introducing extra binary variables to relax and enforce disjunction of constraints. The use MIP solvers would also enable the analysis of specifications involving the use of decimal datatypes. Other solvers could also be used, such as PROLOG-based solvers. While most of the solvers address problems known to be at least NP-Hard, it would be interesting to understand the average case performance for archetypical specifications. This could lead to beneficial analysis results, regardless of the nature of the problem. Before embarking on the use of different types of solvers, the feasibility of the translation to SAT must be assessed. This will be achieved by designing benchmarks using archetypical specifications. Once a good set of benchmarks have been derived for the SAT-based approach, the same set of benchmarks can be reused for MIP solvers and PROLOG-based solvers.
References

Towards the Integration of Visual and Formal Models for GUI Testing *

Ana C. R. Paiva¹, João C. P. Faria¹,², Raul F. A. M. Vidal¹

¹ Engineering Faculty of Porto University, ²INESC Porto,
Rua Dr. Roberto Frias, s/n, 4200-465 Porto, Portugal
{apaiva, jpf, rmvidal}@fe.up.pt

Abstract. This paper presents an approach to diminish the effort required in GUI modelling and test coverage analysis within a model-based GUI testing process. A familiar visual notation – a subset of UML with minor extensions – is used to model the structure, behaviour and usage of GUIs at a high level of abstraction and to describe test adequacy criteria. The GUI visual model is translated automatically to a model-based formal specification language (e.g., Spec#), hiding formal details from the testers. Then, additional behaviour may be added to the formal model to be used as a test oracle. The adequacy of the test cases generated automatically from the formal model is accessed based on the structural coverage of the UML behavioural diagrams.

Keywords: GUI modelling, GUI testing, model-based testing, UML, Spec#.

1 Introduction

Software systems usually feature Graphical User Interfaces (GUIs). They are mediators between systems and users and their quality is a crucial point in the users' decision of using them. GUI testing is a critical activity aimed at finding defects in the GUI or in the overall application, and increasing the confidence in its correctness. Currently it is extremely time-consuming and costly, and very few tools exist to aid in the generation of test cases and in evaluating if the GUI is adequately tested. In fact most of the tools that have been developed to automate GUI testing do not address these two aspects. Capture/replay tools, like WinRunner (www.mercury.com), are the most commonly used for testing GUI applications. They facilitate the construction of test cases through the recording of user interactions into test scripts that can be replayed later, but they still require too much manual effort and postpone testing to the end of the development process, when the GUI is already constructed and functional. They are useful mainly for regression testing. Unit testing frameworks of the XUnit family (e.g., www.junit.org) used together with GUI test libraries (e.g., jemmy.netbeans.org) automate test execution but not test generation. Random input testing tools generate and execute test cases randomly [10].

* Work partially supported by FCT (Portugal) and FEDER (European Union) under contract POSC/EIA/56646/2004.
Recently, model-based approaches for software testing have deserved an increasing attention due to their potential to automate test generation and the increasing adoption of model driven software engineering practices. Some examples of model-based tools developed specifically for GUI testing have been reported [2,8,11]. But the usage of unfamiliar modelling notations, the lack of integrated tool environments, the effort required to construct test-ready models, the test case explosion problem, and the gap between the model and the implementation may barrier the industrial adoption of model-based GUI testing approaches.

In previous work, the authors have developed several extensions to the model-based testing environment Spec Explorer [3] to foster its application for GUI testing and address some of the above issues: techniques and helper libraries for modelling GUIs in Spec# [1]; a GUI Mapping Tool to automate the mapping between the GUI model and the implementation [12]; and a tool to avoid test case explosion taking into account the hierarchical structure of GUIs [13]. Further details can be found in [14]. The results achieved have received significant interest from modellers and testers.

However, the reluctance of modellers and testers to write textual formal specifications that resemble programs is an obstacle to the dissemination of approaches of the above type. Another problem is the lack of support of coverage criteria best adapted for GUI testing, such as coverage of navigation maps.

To address these problems we propose in this paper an approach for model-based GUI testing that aims at combining the strengths of visual modelling (usability) and formal modelling notations (rigor). The basic idea is to provide a familiar visual modelling front-end, based on UML [5], on top of a formal specification language (such as Spec#), with the following goals: diminish the need to write textual specifications; hide as much as possible formalism details from the testers/modellers; use the visual models as the basis for test adequacy/coverage criteria. A subset of UML is selected and several extensions are defined to facilitate GUI modelling and enable the automatic translation to Spec#, where additional behaviour can be added if required. The approach tries to circumvent some UML limitations: difficulty to model several particularities of interactive systems [6], inconsistency problems [4], and the lack of an integrated concrete language for querying and updating the state.

Sections 2 to 5 contain the main contributions of this research work: section 2 describes the overall model-based GUI testing process proposed, integrating visual and formal models; section 3 presents guidelines and stereotypes developed to construct test-ready (and Spec# translatable) GUI models with UML; section 4 describes rules developed to translate the UML behavioural diagrams (namely protocol state machines) into Spec# (namely pre/post-conditions of the methods that trigger state machine transitions); section 5 describes test case generation and coverage analysis of the UML diagrams. Related work is described in section 6; finally, some conclusions and future work are discussed. The Windows Notepad text editor is used as a running example.
2 Overview of the Model-Based GUI Testing Process

Fig. 1 summarizes the activities and artifacts involved in the model-based GUI testing process proposed, which comprises the following steps:

1. **Construction of the visual model** – a set of UML diagrams and additional stereotypes are used to model the usage (via optional use case and activity diagrams), structure (via class diagrams) and behaviour (via state machine diagrams) of the GUI under test.

2. **Visual to formal model translation** – an initial formal model in Spec# is obtained automatically from the UML model according to a set of rules; class diagrams are mapped straightforwardly; state machine diagrams are mapped to pre and post-conditions of methods that model atomic user actions; activity diagrams are translated into methods that model high-level usage scenarios composed of atomic user actions.

3. **Refinement of the formal model** – the Spec# specification resulting from the translation process is completed with method bodies (called model programs in Spec#), in order to obtain an executable model that can be used as a test oracle. The added (explicit) specifications should be consistent with the initial (implicit) specifications, i.e., post-conditions should hold. Consistency may be checked by theorem proving or by model animation during model exploration.

4. **Test case generation and coverage analysis** – test cases are generated automatically by Spec Explorer in two steps: first, a finite state machine (FSM) is extracted from the explicit Spec# specification by bounded exploration of its usually infinite state space; then, test cases are generated from the FSM according to FSM coverage criteria that is usually set to full transition coverage. In the approach proposed, test adequacy is checked by analysing the degree of coverage of the UML behavioural diagrams, which can also be used to limit the exploration.

5. **Model to implementation mapping** – a GUI Mapping Tool [12] allows the tester to interactively relate the abstract user actions defined in the model with concrete

---

**Fig. 1.** Simplified view of the model-based GUI testing process proposed.
actions on physical GUI objects in the application under test, generates a XML file describing the physical GUI objects and methods to simulate the concrete user actions, and binds such methods to the abstract ones for conformity testing.

6. Test case execution – Finally, test cases are executed automatically on the implementation under test and all inconsistencies found are reported.

Steps 4 and 5 and the explicit Spec# model constructed in step 3 are described in [12,13]. The focus of this paper is on the new steps 1 and 2, and the modified step 4.

3 Building translatable visual GUI models with UML

In order to be used as the basis for black-box GUI testing, the GUI model should describe user requirements with enough detail and rigor to allow a person or a machine to decide whether an implementation, as perceived through its GUI, obeys the specification. In addition, it should be easily constructed and analysed. Besides modelling the structure and behaviour of the GUI to support state based testing, we also model the GUI usage to capture high-level requirements and support scenario based testing. Hence, the GUI model is structured as follows:

- **Usage sub-model** (optional) – describes the purpose and typical usages of the application; modelled in UML with use case and activity diagrams;
- **Structure sub-model** – describes the structure of GUI windows; modelled in UML with class diagrams;
- **Behaviour sub-model** – describes inter/intra component behaviour; high-level behaviour is modelled in UML with state machine diagrams; detailed behaviour is modelled by Spec# method bodies.

UML use case diagrams are used optionally to describe the main functionalities and features of the GUI application, as illustrated in Fig. 2. Use cases can be structured as task trees, where higher level tasks are specialized, decomposed (with the «include» UML stereotype) or extended by lower level ones.

![Fig. 2. Example of a partial use case diagram for the Notepad application.](image)
UML activity diagrams are used optionally to detail use cases/tasks in a way that translates directly to test-ready Scenario methods in Spec#. Besides the user steps, they may have parameters that correspond to user inputs, pre/post-conditions (describing use case intent) and assertions, as illustrated in Fig. 3.

![Activity Diagram Example](image)

Fig. 3. Example of a test-ready activity diagram detailing the SaveAs use case from a user perspective (left) and the corresponding Spec# translation (right).

UML class diagrams are used to describe the static structure of the GUI under test at the level of abstraction desired. In most situations, it is appropriate to model top-level windows as objects (instances of classes), while the interactive controls that are contained in those windows are best modelled more abstractly by state variables and/or methods of the containing windows. In the case of the Spec# target language, one can also represent a singleton window by a set of global variables and methods grouped under a common namespace, corresponding to the concept of a module. Several annotations were defined to give special meaning to several UML elements for GUI testing, as shown in Fig. 4. An example of a class diagram is shown in Fig. 5.

The basic behaviour of the several types of windows described in Fig. 4 (including switching the input focus between modeless windows) is handled by a reusable window manager. Information and choice message boxes are on-the-fly windows with a simple structure that need only be represented in state machine diagrams, as illustrated in Fig. 6. A test ready model usually has to include some domain layer model, below the user interface layer. In the case of the Notepad application, it would take care of (non interactive) text file management.
Fig. 4. Stereotypes and other annotations developed for GUI modelling with UML.

<table>
<thead>
<tr>
<th>Annotation</th>
<th>Applies to</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>«ModalWnd»</td>
<td>class, state</td>
<td>Represents a modal window. When open, the other windows of the application are disabled.</td>
</tr>
<tr>
<td>«ModelessWnd»</td>
<td>class, state</td>
<td>Represents a modeless window. When open, it’s still possible to interact with other windows of the same application.</td>
</tr>
<tr>
<td>«InfoMsg»</td>
<td>class, state</td>
<td>Represents an on-the-fly message box that presents information to the user that must be acknowledged.</td>
</tr>
<tr>
<td>«ChoiceMsg»</td>
<td>class, state</td>
<td>Represents an on-the-fly message box that requires the user to choose among a limited range of options.</td>
</tr>
<tr>
<td>«navigate»</td>
<td>association</td>
<td>Models navigation among windows modelled as classes.</td>
</tr>
<tr>
<td>«Action»</td>
<td>method</td>
<td>Describes the effect of an atomic user action (e.g., press a button, enter text) on the GUI state; maps to [Action] annotation in Spec#.</td>
</tr>
<tr>
<td>«Probe»</td>
<td>method</td>
<td>Models the observation of some GUI state from the user eyes (e.g., see the text in a textbox); maps to a [Probe] annotation in Spec#.</td>
</tr>
<tr>
<td>«Property»</td>
<td>method</td>
<td>Models a control with a value that can be set and read by the user; it’s a shorthand for the combination of a state variable, and set (Action) and get (Probe) methods.</td>
</tr>
<tr>
<td>«Scenario»</td>
<td>method</td>
<td>Describes how a user should interact with the GUI to achieve some goal, by a composition of lower level scenarios and atomic actions; maps to a [Scenario] annotation in Spec#.</td>
</tr>
<tr>
<td>{navigationMap}</td>
<td>state machine</td>
<td>Represents the overall navigation map of the application.</td>
</tr>
</tbody>
</table>

Fig. 5. Example of a partial UML class diagram for the GUI of the Notepad application.
UML state machine diagrams are adequate to model the reactive behaviour of GUIs, showing GUI states at different levels of abstraction, the user actions available at each state, their effect on the GUI state, and therefore the possible sequences of user actions. For model-based GUI testing, we advocate the usage of UML protocol state machines [5], instead of behavioural state machines, for the following reasons: they are more abstract (effects are specified implicitly via post-conditions, instead of explicitly via system actions); they promote the separation of concerns between the visual model (pre/post-conditions) and the refinements to introduce in the textual model (executable method bodies); they have the right level of abstraction to express black-box test goals. Each state of a protocol state machine can be formalized by a Boolean condition on the state variables (also called state invariant in [5]). Each transition has a triggering event that, in our case, is the call of a method annotated as Action, representing a user action, and may additionally have pre and post-conditions on state variables and method parameters, according to the syntax [pre]event/[post].

The hierarchical structure of GUIs leads naturally to hierarchical state machines. The overall navigation map of the application can be represented by a top level state machine diagram annotated as (navigationMap), as illustrated in Fig. 6.

Fig. 6. Example of a partial navigation map of the Notepad application.

In multiple window environments, as is the case of GUIs, each state in the navigation map typically represents a situation where a given window (identified by the state name) has the input focus. Transitions represent navigation caused by user actions available in the source window/state of each transition. When one does not want to model the input focus, but only which windows are enabled, orthogonal
(concurrent) states/regions [5] can be used to model multiple modeless windows enabled at the same time. Fig. 6 also illustrates the usage of an intermediate level of abstraction, between the application and the window level, to group together states and transitions related to some use case or task (Finding and Saving composite states). Task reuse via submachine states is also illustrated (SaveBeforeExit submachine state). Junction pseudo-states [5] are used to factor out common parts in transitions.

The internal behaviour of each window shown in the navigation map can be detailed by a lower-level state machine diagram as illustrated in Fig. 7. States in this case represent different window modes or conditions (as identified by state names and formalized by state invariants), distinguished according to display status, user actions available, effect of those actions, test conditions, etc. Transitions are triggered by user actions available in the enclosing window. Orthogonal regions can be used to describe independent state components. For example, in the Notepad main window, the three state components indicated in Fig. 5 (top left) could be described in orthogonal regions. The same event occurrence can fire multiple transitions in orthogonal regions. For example, the TypeText user action can simultaneously cause a transition from HasTextSelected to !HasTextSelected and from !Dirty to Dirty.

![Diagram](image)

Fig. 7. Example of state machine describing internal window behaviour in the Notepad application (left) and partial translation to Spec# (right, simplified).

### 4 Translation to Spec#

A set of rules where developed to translate UML protocol state machines into pre/post-conditions of the Spec# methods that trigger state machine transitions. Some of the rules are presented in Fig. 8, and an application is illustrated in Fig. 7. In some cases (see, e.g., rules R2 and R3), post-conditions may need to refer to the old values of state variables (i.e., the values before method execution). Complex states are reduced by a flattening process (see, e.g., rules R4 and R5). A Boolean simplification post-processing step may be applied (see, e.g., Fig. 7).

The translation of UML activity diagrams into Spec# Scenario methods was already illustrated in Fig. 3. In general, a pre-processing step is required to discover standard structured activities (as illustrated by the dashed boxes in Fig. 3), from which appropriate control structures can then be generated, without jump instructions and control variables whenever possible. The details are outside the scope of this paper.

The translation of the UML class diagrams annotated as described in section 3 into Spec# is reasonably straightforward.
<table>
<thead>
<tr>
<th>Rule</th>
<th>UML protocol state machine</th>
<th>Translation to Spec# / Reduction</th>
</tr>
</thead>
</table>
| **R1.** Simple transition | $S_1$ [cond1(svars)] $\xrightarrow{\text{pre(svars,params)} \land \text{m(params)}} S_2$ [cond2(svars)] | [Action] $m$(params) requires cond1(svars) && pre(svars,params); ensures post(svars,params) && cond2(svars); 
{ ... } (*svars* – state variables) |
| **R2.** Multiple transitions with the same trigger | $S_1$ [p1]m1/[q1] $\xrightarrow{\text{p2} \land \text{m1} \lor \text{p3} \land \text{m2}} S_2$ $\xrightarrow{\text{p4} \land \text{m3}} S_3$ $\xrightarrow{\text{p5} \land \text{m4}} \ldots S_n$ | [Action] $m_1$(params) requires (S1 && p1) || (S3 && p2) || ...;
ensures (S1 && p1 && q1 && S2) && (S3 && p2 && q2 && S4) && ...; |
| **R3.** Junction pseudo-state (branch with "else" case) | $S_1$ [p1]m1/[q1] $\xrightarrow{\text{p2} \land \text{m1} \lor \text{p3} \land \text{m2} \lor \text{p4} \land \text{m3} \lor \ldots}$ $\xrightarrow{\text{p5} \land \text{m4}} S_2$ $\xrightarrow{\text{p6} \land \text{m5}} S_3$ $\xrightarrow{\text{p7} \land \text{m6}} S_4$ $\xrightarrow{\text{p8} \land \text{m7}} \ldots S_n$ | [Action] $m_1$(params) requires S1 && p1;
ensures q1 && ($p_2 ? q_2 \land S_2 : p_3 ? q_3 \land S_3 : \ldots ? q_n \land S_n$); (*?* is the C ternary operator) |
| **R4.** Composite state (S) | $S$ $\xrightarrow{[q]} S_1$ $\xrightarrow{[p_2]m_2/[q_2]} S_2$ $\xrightarrow{[p_3]m_3/[q_3]} S_3$ $\xrightarrow{[p_4]} S_4$ | [Action] $m_3$(params) requires S1 && p1;
ensures (S1 && p1 && q1 && S2) && (S2 && p2 && q2 && S3) && (S3 && p3 && q3 && S4) && ...;
ensures (S1 && p1 && q1 && S2) && (S2 && p2 && q2 && S3) && (S3 && p3 && q3 && S4) && ...; |
| **R5.** Submachine states (S1 ... Sn) | $S$ $\xrightarrow{[q]} S_1$ $\xrightarrow{[p_1]} S_1$ $\xrightarrow{[p_2]} S_2$ $\xrightarrow{[p_3]} S_3$ $\xrightarrow{[p_4]} S_4$ $\xrightarrow{[p_5]} S_5$ $\ldots S_n$ $\xrightarrow{[q_n]} S_n$ | [Action] $m_3$(params) requires S1 && p1;
ensures (S1 && p1 && q1 && S2) && (S2 && p2 && q2 && S3) && (S3 && p3 && q3 && S4) && ...;
ensures (S1 && p1 && q1 && S2) && (S2 && p2 && q2 && S3) && (S3 && p3 && q3 && S4) && ...; |

**Fig. 8.** Translation rules from UML protocol state machines into Spec#. State and pre/post-conditions are abbreviated after rule R1. Due to space limitations, rules for other features (e.g., fork, join, entry, exit and merge pseudo-states) are omitted.
5 Test Case Generation and Coverage Analysis

With Spec Explorer, the exploration of a Spec# model to generate a finite test suite (see section 2) is based on parameter domain values provided by the tester, but there is no feedback mechanism to evaluate the quality of the test cases obtained based on appropriate GUI test adequacy criteria. To overcome this problem, we propose the structural coverage of the UML state machine model as a test adequacy criterion. To report the degree of coverage achieved and bound the exploration, the exploration process is extended as follows. Every time an action is explored with actual parameter values, it is checked if there is a corresponding transition (or set of transitions from orthogonal states) in the UML model (by evaluating their pre/post-conditions and source/target state conditions with the actual values of parameters and state variables) and, in that case, their colour is changed; otherwise, a consistency warning (if the state does not change) or error is reported. The exploration can be stopped as soon as the specified degree of coverage of the UML state machine model is achieved. At the end of the exploration, the tester knows if the coverage criteria defined were satisfied and can use the information provided to refine the domain values or the models.

A complementary technique to generate test cases is based on scenarios' coverage. In fact, the methods annotated as Scenario, describing typical ways (but not all the possible ways) of using the system, can be used as parameterized test cases for scenario based testing. Symbolic execution and constraint solving may be used to generate a set of parameter values that guarantee exercising those scenarios to a certain degree of structural coverage [16]. Scenario based testing leads to a small number of test cases, that are more adapted to test GUIs usability and more useful as acceptance tests, but does not cover the GUI functionality to the same extent as the above state based testing approach can potentially do. Scenarios can also be constructed to exercise behaviour that is difficult to cover with state based testing.

6 Related Work

There are few examples of model-based GUI testing tools. The main characteristics of two of them will be presented next.

IDATG [2] (Integrated Design and Automated Test Case Generation Environment) provides an editor to assist and facilitate the construction of the GUI specification as atomic user actions and as (task-oriented) test scenarios. Test cases may be generated to cover the functionality of the GUI from the former model and to check the usability of the GUI from the latter. Although it may be more pleasant to construct the model using an editor, IDATG does not provide a way to model different features of the GUI, i.e., different views of the lower level model, and to assure consistency among those views. In addition, IDATG does not provide support for test case execution, which requires a change of environment, for instance, using WinRunner.

GUITAR (GUI Testing Framework) provides a dynamic reverse engineering process to construct the model of already existing GUIs as a way to reduce the time and effort needed in their construction. The GUI model comprises an event flow graph to model intra-component behaviour and an integration tree to model
inter-component behaviour [7]. These models can be viewed graphically. However, they are not editable and cannot be constructed manually. As a consequence, its industrial applicability beyond regression testing is questionable.

There are also examples of graphical notations for modelling GUIs in the context of user interface design. Several notations are based on UML and its extension mechanisms. E.g., UMLi (UML for Interactive Applications) is an extension to UML aiming to integrate the design of applications and their user interfaces [15]. It introduces a graphical notation for modelling presentation aspects, and extends activity diagrams to describe collaboration between interaction and domain objects. Another example is the UML profile defined in the Wisdom approach [9]. However, most of these extensions are not sufficient to describe GUIs with the detail and rigour required by model-based testing tools.

7 Conclusions and Future Work

We have presented an approach to foster the adoption of model-based GUI testing approaches by diminishing the effort required in GUI modelling and test coverage analysis. It consists of a familiar UML-based visual notation, a translation mechanism into Spec#, and a test adequacy analysis technique. The visual notation is used to model GUIs at a high-level of abstraction and at the same time to describe test adequacy criteria. The translation mechanism aims to hide formalism details as much as possible. The adequacy of the test cases generated is accessed and reported graphically to the user on the visual models.

We are currently extending the model-based testing tools developed in previous work to support the techniques described in this paper. The prototype under development is able to manipulate UML models represented in the XML Metadata Interchange (XMI) format (www.omg.org/technology/documents/formal/xmi.htm).

As future work, we intend to:
− extend the tools with round-trip engineering capabilities, following the principle that the UML diagrams are partial views over the formal model;
− explore other visual behaviour modelling techniques, such as UML behavioural state machines, in order to completely hide the Spec# model at the price of a more detailed visual model (with procedural actions on transitions), and/or produce less coupled models (with the exchange of signals between concurrent state machines) at the price of a more complex execution model;
− reduce even further the modelling effort for already existing GUIs by extracting a partial GUI model by a reverse engineering process;
− use temporal logic to express additional test goals and model-checking to generate test cases, taking advantage of the IVY platform (www.di.uminho.pt/ivy);
− validate the approach in an industrial environment.
References

Testing Planning Domains (without Model Checkers)

Franco Raimondi\textsuperscript{1} and Charles Pecheur\textsuperscript{2} and Guillaume Brat\textsuperscript{3}

\textsuperscript{1} Department of Computer Science, University College London, London, UK
\textsuperscript{2} Dept. INGI, Université catholique de Louvain, Louvain-la-neuve, Belgium
\textsuperscript{3} RIACS/NASA Ames Research Center, Moffett Field, CA, USA

\textbf{Abstract.} We address the problem of verifying planning domains as used in model-based planning, for example in space missions. We propose a methodology for testing flight rules of planning domains which is self-contained, in the sense that flight rules are verified using a planner and no external tools are required. We review and analyse coverage conditions for requirements-based testing, and we reason in detail on "Unique First Cause" (UFC) coverage for test suites. We characterise flight rules using patterns, encoded using LTL, and we provide UFC coverage for them. We then present a translation of LTL formulae into planning goals, and illustrate our approach on a case study.

1 Introduction

The NASA rovers Spirit and Opportunity [1, 2], exploring the surface of Mars since January 2004, are an example of complex systems, featuring significant autonomous capabilities, that can be built using current software and hardware technologies. Due to the complexity of such systems and in order to avoid economic losses and mission failures, there is a growing interest in tools and methodologies to perform formal verification for this kind of autonomous applications. In this paper we are concerned with the problem of verifying planning domains. In the case of the Mars rovers, plans are generated using the Europa 2 planner [3] to satisfy some scientific and positioning goals. Then, the plans are manually checked (against a set of requirements called flight rules) before being uploaded to the rover. The generation and verification have to be done before the next Mars daytime cycle. The methodology we propose to verify planning domains would speed up the verification phase and help ensure that flight rules are not violated.

Verification of planning domains has been investigated in the past, for instance in [4, 5]. The solutions proposed by these authors consist in the translation of the planning domain into the input language of some model checker. The main limitation of these approaches is the limited size of the domains that can be translated and the problematic correspondence between languages for planners and languages for model checkers. In this paper we suggest a different approach:

\textsuperscript{*} The author acknowledge support from the European IST project PLASTIC and from Mission Critical Technologies, NASA Ames Research Center
we propose to translate the problem of verification of planning domains into a planning problem. Such an approach has the advantage that no external tools are required, because the actual planner can be used to perform verification. Specifically, we suggest to proceed as summarised in Figure 1: given as input a planning domain and a set of flight rules (this is normally provided as a text document), in step 1 we encode flight rules as LTL specifications. In the second step we derive test cases from the specifications; we employ a revised notion of requirements-based testing, using an approach similar to [6–8]. In the third step we convert the test cases generated into planning goals. The actual tests are performed by “enriching” the original planning model with the new set of goals, using the planner for which the domain was designed.

![Figure 1. From flight rules to planning goals (overview).](image-url)

The rest of the paper is organised as follows. In Section 2 we review the notion of coverage for requirements-based testing. We present the planning problem in Section 3. A motivational example is introduced in Section 4. We show how flight rules can be encoded as temporal formulae in Section 5, and how these can be translated into planning goals in Section 6, using the example provided. We conclude in Section 7.

2 Testing and requirements-based testing

Various metrics exist to quantify the coverage of test suites [9], particularly for structural testing. Coverage metrics for functional testing can be defined as well when functional requirements are provided formally, for instance using temporal formulae. Previous work in this direction include [8, 7, 6]. In this section we briefly review MC/DC (structural) coverage, and then we reason on a metric for functional testing by extending some concepts from [6].

2.1 MC/DC Coverage

In the scope of this paper, we use a metric inspired by the popular Modified Condition/Decision Coverage (MC/DC) [10]. In particular, MC/DC coverage is required for the most critical categories of avionic software [11] and its extensions can be employed in specification-based testing (see below). MC/DC coverage is defined in terms of the Boolean decisions in the program, such as test expressions in if and while statements, and the elementary conditions (i.e. Boolean terms) that compose them. A test suite is said to achieve MC/DC if its execution ensures...
that: (1) Every basic condition in any decision has taken on all possible outcomes at least once. (2) Each basic condition has been shown to independently affect the decision’s outcome.

As an example, the program fragment if (a || b) { ... } contains the decision \( c \equiv (a \lor b) \) with conditions \( a \) and \( b \). MC/DC is achieved if this decision is exercised with the following three valuations:

(1) \( a = \top, b = \bot, c = \top \); (2) \( a = \bot, b = \top, c = \top \); (3) \( a = \bot, b = \bot, c = \bot \).

Indeed, evaluations 1 and 3 only differ in \( a \), showing cases where \( a \) independently affects the outcome of \( c \), respectively in a positive and negative way. The same argument applies to evaluations 2 and 3 for \( b \).

The MC/DC requirements for each condition can be captured by a pair of Boolean formulae, called trap formulae, capturing those valuations in which the condition is shown to positively and negatively affect the decision in which it occurs (also called the positive and the negative test cases). Coverage is achieved by building test cases that exercise the condition in states which satisfy each trap formula. In the example above, the trap formulae for condition \( a \) in \( c \) are \( a \land \neg b \) and \( \neg a \land \neg b \).

### 2.2 UFC Coverage

As mentioned above, if functional specifications are expressed in a formal framework, then functional testing can be measured in terms of coverage criteria similar to structural testing, but applied to the specifications rather than the implementation. This is the idea behind requirements-based testing, as investigated in [6] and in [7]. In particular, [6] defines the notion of Unique First Cause coverage (UFC), which extends ideas from MC/DC to requirements-based testing.

UFC coverage is defined with respect to functional properties that executions of a system must satisfy, and to the atomic conditions occurring in these properties. As illustrated in Section 5, these properties are often expressed by means of temporal formulae, for instance using the logic LTL (we refer to [12] for more details). A test suite achieves UFC coverage of a set of requirements expressed as temporal formulae, if: (1) Every basic condition in any formula has taken on all possible outcomes at least once. (2) Each basic condition has been shown to affect the formula’s outcome as the unique first cause.

Following [6], a condition \( a \) is the unique first cause (UFC) for \( \varphi \) along a path \( \pi \) if, in the first state along \( \pi \) in which \( \varphi \) is satisfied, it is satisfied because of \( a \). This can be formalised as follows. Let \( \pi \) be a a path and \( X \) a set of atomic conditions; we denote by \( \pi(X) \) the sequence of truth values of the atomic conditions in \( X \) along \( \pi \) (also called the projection of \( \pi \) over \( X \)).

**Definition 1 (\( \alpha \)-variant).** Let \( AC(\varphi) \) be the set of occurrences of atomic conditions in a linear temporal formula \( \varphi \).\(^5\) Given \( a \in AC(\varphi) \) and a path \( \pi \), an \( \alpha \)-variant of \( \pi \) w.r.t. \( \varphi \) is a path \( \pi' \) such that \( \pi(AC(\varphi) - \{a\}) = \pi'(AC(\varphi) - \{a\}) \).

\(^4\) We use \( \top \) and \( \bot \) to denote Boolean true and false.

\(^5\) Note that different occurrences of the same condition \( a \) are considered distinct. This poses technical difficulties with multiple occurrences. This is a known issue in the MC/DC context too.
Definition 2 (UFC coverage). Given a linear temporal formula \( \varphi \), a condition \( a \in AC(\varphi) \) is the unique first cause (UFC) for \( \varphi \) along a \( \pi \), or equivalently, \( \pi \) is an adequate (UFC positive) test case for \( a \) in \( \varphi \), if \( \pi \models \varphi \) and there is an \( a \)-variant \( \pi' \) of \( \pi \) such that \( \pi' \nvdash \varphi \).

When \( \varphi \) is a LTL formula, one can build a trap formula \( ufc(a, \varphi) \), which is a temporal formula characterising adequate test cases for \( a \) in \( \varphi \), i.e., paths on which \( a \) is UFC for \( \varphi \).\(^6\) \( ufc(a, \varphi) \) is defined by induction on \( \varphi \). For example, given \( a \in AC(\varphi_a) \):

\[
ufc(a, a) = a; \quad ufc(a, \neg a) = \neg a \\
ufc(a, \varphi_a \lor \varphi_b) = \ufc(a, \varphi_a) \land \neg \varphi_b \\
ufc(a, F \varphi_a) = (\neg \varphi_a) \lor \ufc(a, \varphi_a) \\
ufc(a, G \varphi_a) = \varphi_a \lor (\ufc(a, \varphi_a) \land G \varphi_a)
\]

A complete definition is found in [6], and as a refined version later in this section. This characterisation of test cases for LTL only applies to complete, infinite paths. Realistic testing practices, however, are inherently limited to finite, truncated paths. In this context, the test case coverage criteria need to be refined further. Consider, for instance, the formula \( \varphi = G (a \lor b) \) expressing the requirement that either \( a \) or \( b \) must hold at any time. According to the definition above, an adequate test case for \( a \) in \( \varphi \) must satisfy

\[
ufc(a, \varphi) = (a \lor b) \lor ((a \land \neg b) \land G (a \lor b))
\]

A concrete, finite test case \( \pi_f \) may reach a point where \( a \land \neg b \), showing evidence that \( a \) may contribute to making \( \varphi \) true. However, there is no guarantee that this \( \pi_f \) is indeed a prefix of a \( \pi \) that satisfies \( \varphi \), that is, that \( a \lor b \) can hold indefinitely beyond the end of \( \pi_f \). Such a finite prefix is defined as a weak test case for \( a \) in \( G (a \lor b) \).

A comprehensive treatment of temporal logic on truncated paths is found in [13], where strong and weak variants of semantic relations on finite prefixes are defined (\( \pi_f \models^+ \varphi \) and \( \pi_f \models^- \varphi \), respectively), where \( \pi_f \models^ \varphi \) iff \( \pi_f \models^+ \varphi \) and \( \neg \varphi \). Intuitively, \( \pi_f \models^ \varphi \) iff \( \pi_f \) “carries all necessary evidence for” \( \varphi \), whereas \( \pi_f \models^- \varphi \) iff it “carries no evidence against” \( \varphi \). In particular, if \( \pi_f \models^+ \varphi \), then for any (non-truncated) \( \pi \) extending \( \pi_f \) we have \( \pi \models \varphi \). Furthermore, \([6]\) defines strengthening and weakening transformations \( [\varphi]^+ \) and \( [\varphi]^− \) such that \( \pi_f \models^+ \varphi \) iff \( \pi_f \models [\varphi]^+ \).\(^7\) In essence, \( [\varphi]^+ \) converts weak untils to strong untils, and vice-versa for \( [\varphi]^− \); in particular, \( [G \varphi]^+ = \perp \) and \( [F \varphi]^− = \top \).

On this basis, \([6]\) adapts the notion of UFC coverage by requiring that a (finite) test \( \pi_f \) case for \( a \) in \( \varphi \) satisfies \( \varphi \) according to the standard semantics up

\(^6\) [6] uses a different notation \( \varphi^+ \) for the set of (positive) trap formulae for conditions in \( \varphi \), that is, \( \varphi^+ = \{ ufc(a, \varphi) \mid a \text{ occurs in } \varphi \} \). The notation \( \varphi^− = (\neg \varphi)^+ \) is also defined. We do not use these notations here to avoid confusion with strong/weak semantic variants \( \models^+ \) and \( \models^- \) (see below).

\(^7\) Denoted \( strong[\varphi] \) and \( weak[\varphi] \) in [6], using LTL semantics extended to finite traces as in [13].
to the point where the effect of $a$ is shown, and according to the weak semantics thereafter. For example, the trap formula for $\varphi = G (a \Rightarrow F b)$ becomes

$$ufc(a, \varphi) = (a \Rightarrow F b) \cup ((\neg a \land \neg F b) \land [G (a \Rightarrow F b)]^-)$$

$$= (a \Rightarrow F b) \cup (\neg a \land \neg F b)$$

since $[G (a \Rightarrow F b)]^-$ reduces to $\top$. This also illustrates a lack of uniformity in the approach: the weakening cancels some of the obligations on $F b$, but not all, although the truncation may equally prevent all of them from being reached. Instead, in this paper we keep both weak and strong interpretations and apply them uniformly, obtaining two refined variants of UFC coverage.

**Definition 3 (Strong/weak UFC coverage).** Given a linear temporal formula $\varphi$, a condition $a \in AC(\varphi)$ is the strong (resp. weak) unique first cause for $\varphi$ along a finite path $\pi_f$, or equivalently, $\pi_f$ is an adequate strong (resp. weak) test case for $a$ in $\varphi$, if $\pi_f \models^+ \varphi$ (resp. $\models^-$) and there is an $a$-variant $\pi'_f$ of $\pi_f$ such that $\pi'_f \not\models^+ \varphi$ (resp. $\models^-$).

As an example, the prefix in Figure 2 is, at the same time, a strong test case for $a$ in $F (a \land \neg b)$ and a weak test case for $a$ in $G (a \lor b)$. Observe that, consistently with the discussion above, no finite prefix can strongly test a formula such as $G a$, whose negation contains eventualities. We then refine $ufc(a, \varphi)$ into strong and weak variants $ufc^+$ and $ufc^-$, such that $\pi_f \models ufc^\pm(a, \varphi)$ iff $\pi_f$ is a strong/weak test case for $a$ in $\varphi$. These are jointly defined as follows, given $a \in AC(\varphi_a)$ and $b \in AC(\varphi_b)$.

$$ufc^+(a, a) = a ; ufc^+(a, \neg a) = \neg a$$

$$ufc^+(a, \varphi_a \land \varphi_b) = ufc^+(a, \varphi_a) \land [\varphi_b]^\pm$$

$$ufc^+(a, \varphi_a \lor \varphi_b) = ufc^+(a, \varphi_a) \lor [\neg \varphi_b]^\pm$$

$$ufc^+(a, X \varphi_a) = X ufc^+(a, \varphi_a)$$

$$ufc^+(a, F \varphi_a) = [\neg \varphi_a]^\pm \cup ufc^+(a, \varphi_a)$$

$$ufc^+(a, G \varphi_a) = [\varphi_a]^\pm \cup (ufc^+(a, \varphi_a) \land [G \varphi_a]^\pm)$$

$$ufc^+(a, \varphi_a \lor \varphi_b) = [\varphi_a \land \neg \varphi_b]^\pm \cup (ufc^+(a, \varphi_a) \land [\neg \varphi_b]^\pm \lor [\varphi_a \lor \varphi_b]^\pm)$$

$$ufc^+(b, \varphi_a \lor \varphi_b) = [\varphi_a \land \neg \varphi_b]^\pm \cup ufc^+(b, \varphi_b)$$

$$ufc^+(a, \varphi_a \lor \varphi_b) = [\varphi_a \land \neg \varphi_b]^\pm \cup (ufc^+(a, \varphi_a) \land [\neg \varphi_b]^\pm \lor [\varphi_a \lor \varphi_b]^\pm)$$

$$ufc^+(b, \varphi_a \lor \varphi_b) = [\varphi_a \land \neg \varphi_b]^\pm \cup ufc^+(b, \varphi_b)$$

---

This definition covers all cases, by pushing negations down to atoms and by symmetry of Boolean operators. Cases for $F$ and $G$ could be derived from $U$ and $W$. 

Fig. 2. Strong test case for $a$ in $F (a \land \neg b)$ and weak test case for $a$ in $G (a \lor b)$. 

---

106
Specifically, sub-terms in $G$ and $W$ are strengthened to $\bot$ and $U$ in the $+$-cases; in particular, $ufc^+(a, G \varphi)$ boils down to $\bot$ for any $\varphi$ (not a tautology), reflecting the fact that no adequate strong (finite) test case exists for $G \varphi$. Given an atomic condition $a$ appearing in a formula $\varphi$ and an execution model $M$, if there is a strong test case $\pi_f \models ufc^+(a, \varphi)$ in the traces of $M$, then $\pi_f$ shows that $a$ can necessarily positively affect $\varphi$, in the sense that any extension $\pi$ of $\pi_f$ indeed satisfies $\varphi$. On the other hand, if $\pi_f \models ufc^-(a, \varphi)$, then $\pi_f$ only shows that $a$ can possibly positively affect $\varphi$, in the sense that there is no guarantee that this prefix can be completed to a full path of $M$ that satisfies $\varphi$. It is also possible that there is no $\pi_f$ in $M$ for which $\pi_f \models ufc^\pm(a, \varphi)$: if $\varphi$ is a positive (desired) property, then this means that $a$ is a vacuous condition in $\varphi$ w.r.t. $M$ [7]; if $\varphi$ is a negative (forbidden) property, then it confirms that this particular case of $\varphi$ cannot happen, which is the desired result. A test fails if it is possible to find a path $\pi_f$ in $M$ such that $\pi_f \models ufc^\pm(a, \varphi)$, where $\varphi$ is a negative property.

### 3 Planning

[...]

Planning can be considered a process of generating descriptions of how to operate some system to accomplish something. The resulting descriptions are called plans, and the desired accomplishments are called goals. In order to generate plans for a given system a model of how the system works must be given." [3]

Traditionally, in the description of a system there is a distinction between states and actions (see, for instance, the STRIPS planner and its associated language [14]). In this paper, however, we take a different approach and we consider the Europa 2 planner [3]; Europa 2 produces finite horizon, deterministic plans. The key concept of Europa 2 is the one of tokens, i.e., a temporally scoped (true) fact. For instance, a printer being ready between the time $t = 2$ and $t = 5$ is represented using a token $\textit{ready}$ (see the first token in Figure 3).

Tokens may represent states of a single object in the system, and are sometimes mutually exclusive. A timeline is a structure where sequences of tokens appear contiguously. For instance, the state of a printer can be described by a sequence of tokens, as in Figure 3 (therefore, state is a timeline). In this example, the syntax $[2]$ denotes the time instant $t = 2$, while $[3 - 5]$ denotes a value of $t$ between 3 and 5.

Europa 2 allows for the expression of various relations between tokens, based on Allen’s interval logic [15]. Allen’s interval logic includes 13 possible relations between a pair of tokens: meets/met by, before/after, equals,
A planning problem is formulated in Europa 2 as a set of tokens (possibly belonging in a timeline), a set of rules expressed using the relations presented above, and a set of goals: these elements define a so-called partial plan, which is refined by Europa 2 into a complete plan, i.e., a plan where all tokens are active and no unbound variables exist. We refer to [3] and references therein for further details about the theoretical foundations of Europa 2. The input files of Europa 2 are expressed using the New Domain Description Language (NDDL, pronounced “noodle”). NDDL is a (restricted) object-oriented, Java-like language; a simple example of a NDDL file (without goals) is given in Figure 4.

Fig. 4. A very simple NDDL file.

4 A concrete example: the Rover scenario

This section presents a concrete case study which will be used in the remainder of the paper to exemplify our methodology: an autonomous rover. A rover contains three main components: a navigator to manage the location and the movement of the rover; an instrument to perform scientific analysis; a commander, receiving instructions from scientists and directing the rover’s operations. Each of these components is mapped into a class (respectively: Navigator, Instrument, and Commands); each of these classes is a timeline. Additionally, the domain contains two classes Location and Path to represent physical locations and paths between locations. Each of the timelines contain the following tokens:

- **Navigator**: At, Going (the rover is either at a location, or going between locations).
- **Instrument**: stow, unstow, stowed, place, takesample (the instrument can be stowed, can be in the state of being stowed or unstowed, can be placed on a target, or it can take a sample).
- **Commands**: takesample, phonehome, phonelander (the rover can be instructed to take a sample, and it can communicate either with a lander, or directly with the Earth).

An extract of the NDDL code for this example is presented in Figure 5, where the NDDL code corresponding to a part of the declaration of the Navigator timeline is shown (notice the similarities with the syntax of Java). See the comments appearing in Figure 5 for more details. The class Timeline (not shown...
class Location {
    string name; int x; int y;
    Location(string _name, int _x, int _y) {
        name = _name; x = _x; y = _y;
    }
}

// Navigator is a Timeline
class Navigator extends Timeline {
    predicate At{
        Location location;
    }
    predicate Going{
        Location from; Location to;
        neq(from, to);
        // This is a rule: it prevents rover from going from a location
        // straight back to that location.
    }
}

// This is a rule for the token At
Navigator::At{
    met_by(object.Going from);// Allen’s relation: each At is met_by
    eq(from.to, location); // next Going token starts at this location
    meets(object.Going to); // Allen’s relation: each At meets
    eq(to.from, location); // previous Going token ends at this location
}

// A possible goal
goal(Commands.TakeSample sample); sample.start.specify(50);

Fig. 5. Excerpt from the NDDL file for the rover example.

in the example) is a super-class containing two variables start and end (notice
that in NDDL all variables and methods are public). A possible goal for this
scenario is presented at the end of the code in Figure 5: the goal is to begin a
sample of rock4 at time 50.

When a NDDL file encoding the scenario and the goals is passed to Europa
2, the planner generates a plan, similarly to the one presented in Figure 6.

Fig. 6. Generated plan.

5 Flight rules

Flight Rules are requirements that must be satisfied in every execution. Typi-
cally, flight rules are a plain text document which accompanies the description
of scenarios. For instance, a flight rule for the example presented in Section 4
is “all Instruments must be stowed when moving”. The majority of flight rules
falls into one of the temporal patterns defined in [16] and thus can be encoded
by means of an LTL formula. For instance, the flight rule above can be encoded
as $\varphi = G(p \rightarrow q) = G(\neg p \lor q)$, where $p$ = moving and $q$ = stowed.
5.1 Coverage sets for flight rules

As flight rules can be encoded as LTL formulae, the methodology presented in Section 2 can be applied to generate a set of test cases for flight rules with coverage guarantees. As an example, we consider the flight rule presented above, namely \( \varphi = G(\neg p \lor q) \) (where \( p \) = moving and \( q \) = stowed). Being a safety formula, we can only have weak evidences for the positive test cases (see Section 2.2) because the planner can only generate finite executions. More in detail, we have the following three test cases:

1. \( ufc^- (q, \varphi) = ((\neg p \land q)U(\neg p \land \neg q)); \)
2. \( ufc^- (p, \varphi) = ((\neg p \land q)U(p \land q \land \varphi)); \)
3. \( ufc^{+/-} (p, \varphi) = ufc^{+/-} (q, \varphi) = ((\neg p \lor q)U(p \land \neg q)). \)

The first positive test case tests the true value of the whole formula caused by the proposition \( q \) (i.e., stowed); the second test case tests the proposition \( p \). There is only one test case for false value of the formula and it is contributed by both propositions; notice that this test case is the same for weak and for strong evidence.

A similar exercise could be repeated for all the flight rules appearing in the specification for any given scenarios. Using the methodology presented above to compute test cases with UFC coverage guarantees would be an improvement per se with respect to the current testing methodologies (currently, test cases for Europa 2 are generated manually and without coverage guarantees). But our approach can be refined further, to the benefit of plan developers: in the next section we present how the execution of tests can be automated by translating temporal formulae into planning goals.

6 From temporal formulae to planning goals

The key idea of this section is to translate LTL formulae encoding test cases into planning goals. This is achieved by building a parse tree of the formula and by associating a timeline to each node of the parse tree.

We present the details of this methodology using the first positive test case for the scenario presented in Section 4, namely, for the formula \( (\neg p \land q)U(\neg p \land \neg q) \), where \( p \) = moving and \( q \) = stowed.

We start with proposition \( p \), which is true whenever the rover is moving. We define a new timeline \( \text{prop-p} \) containing the two tokens \( \text{TRUE} \) and \( \text{FALSE} \): \( \text{TRUE} \) token of \( \text{prop-p} \) is the case when the token \text{Going of Navigator} holds, and token \( \text{FALSE} \) of the timeline \( \text{prop-p} \) holds when \( \text{TRUE} \) does not hold. These requirements are translated into the NDDL code represented in Figure 7 (top part). The negation of proposition \( p \) is defined as a new timeline \( \text{prop-not-p} \) composed by the two tokens \( \text{TRUE} \) and \( \text{FALSE} \) and by the rules presented in Figure 7 (bottom part).

Proposition \( q \) (representing the fact that instruments are stowed) and its negation are defined in a similar way as new timelines. The conjunction of two propositions is encoded as a new timeline with four tokens representing the
possible truth values of the two conjuncts. The scope of each token is defined using the two Allen’s relations contains and contained_by. The truth value of the whole conjunction is obtained using a third timeline with two tokens only (TRUE and FALSE). The NDDL code corresponding to the conjunction of two propositions is available from the authors upon request. We are now in a position to test the formula $\varphi = (\neg p \land q) U (\neg p \land \neg q)$. For simplicity, let $\varphi = A U B$, where $A$ and $B$ are encoded using the timelines prop-A and prop-B respectively.

The LTL proposition $\varphi$ holds in the model iff the following goal can be satisfied:

$$
goal(prop-A.TRUE); goal(prop-B.TRUE); eq(prop-A.start,0);
contains_end(prop-B.TRUE,prop-A.TRUE);$$

Intuitively, the goal above states that proposition $A$ has to be true at the beginning of the run (eq(prop-A.start,0)) and that $B$ becomes true before the end of TRUE of $A$ (contains_end(prop-B.TRUE,prop-A.TRUE)). The additional NDDL code presented above is added to the original NDDL code for the scenario (notice that, in doing so, the instrumentation process cannot introduce bugs in the original model). The new “enriched” NDDL code is passed to Europa 2 for plan generation. If a plan can be obtained with the additional constraints for the positive test case, the test is passed successfully. Figure 8 depicts Europa 2 output for the first test case. Notice the additional timelines for the propositions Boolean propositions (compare with Figure 6). This plan illustrates an execution where the atomic condition $q$ (stowed) is the unique first cause. This exercise can be repeated for the second positive test case, which is passed, and for the negative test case. As expected, no plan can be generated for the negative test case.

**6.1 Discussion**

While the scenario presented above is not as rich as real production and mission environments, it is nevertheless more complex than the biggest examples that could be
analysed using translations into model checkers [5, 4]. We have run our tests on a standard machine and the introduction of the new timelines did not affect the performance of the planner for positive test cases. This result was expected, as a domain with additional constraints should be “easier” to solve than a less constrained domain: the introduction of the new timelines seems to balance this benefit. Negative test cases, however, require more computational power because of the backtracking involved in over-constrained domains. The planner eventually fails on negative test cases in around 10 minutes for the rover example, while it is able to produce a result in less than 30 seconds for positive test cases. Even though our aim in this paper is mainly to provide feasible coverage guarantees for test suites of planning domains and we are not concerned with performance issues, nevertheless we consider our preliminary results encouraging.

7 Conclusion

Traditionally, the problem of verifying planning domains has been approached by translating the planning domain into an appropriate formalism for a model checker, where verification can be performed either in a direct way, or by generating test cases, with the exception of [17] where a planning techniques are suggested for the generation of tests. This latter work differs from ours in that different coverage conditions are considered, and tests are not generated from flight rules (i.e., temporal specifications). Some issues remain to be investigated. For instance, we do not have a methodology to deal with the translation of nested temporal operators into planning goals (but we did not find nested temporal operators in the flight rules analysed). We are currently working on this issue and on a software tool to automate the methodology: we are implementing a parser from temporal patterns encoding flight rules to LTL trap formulae using the definitions in Section 2.2, and finally to NDDL code.

References
