EPTCS 141

Proceedings of the
Ninth Workshop on
Model-Based Testing

Grenoble, France, 6 April 2014

Edited by: Holger Schlingloff and Alexander K. Petrenko
Table of Contents

Table of Contents ................................................................. i

Preface ................................................................. ii
   Alexander K. Petrenko and Holger Schlingloff

Using Formal Specifications to Support Model Based Testing ASDSpec: A Tool Combining the
Best of Two Techniques .............................................................. 1
   A.P. van der Meer, R. Kherrazi and M. Hamilton

Verifying Web Applications: From Business Level Specifications to Automated Model-Based
Testing .............................................................. 14
   Christian Colombo, Mark Micallef and Mark Scerri

Coverage Criteria for Model-Based Testing using Property Patterns ...................... 29
   Kalou Cabrera Castillos, Frédéric Dadeau and Jacques Julliand

Spinal Test Suites for Software Product Lines .............................................................. 44
   Harsh Beohar and Mohammad Reza Mousavi

Generating Complete and Finite Test Suite for ioco: Is It Possible? ......................... 56
   Adenilso Simao and Alexandre Petrenko
Preface

This volume contains the proceedings of the Ninth Workshop on Model-Based Testing (MBT 2014), which was held in Grenoble, France on April 6, 2014 as a satellite workshop of the European Joint Conferences on Theory and Practice of Software (ETAPS 2014).

The first workshop on Model-Based Testing (MBT) in this series took place in 2004, in Barcelona. At that time model-based testing already had become a hot topic, but MBT 2004 was the first event devoted exclusively to this domain. Since that time the area has generated enormous scientific and industrial interest, and today there are several other workshops and conferences on software and systems design, modelling and quality assurance, covering also model based testing. In a way, MBT has become mainstream, and in various domains the technology has matured to an industrially established practice. Still, the MBT series of workshops offers a unique opportunity to share new technological and foundational ideas particular in this area, and to bring together researchers and users of model-based testing to discuss the state of the theory, applications, tools, and industrialization.

Model-based testing has become one of the most powerful system analysis methods, where the range of possible applications is still growing. Similar as in previous years, we see the following main directions of development:

- Integration of model-based testing techniques with various other analysis techniques; in particular, integration with formal development methods and verification tools;
- Application of the technology in the certification of safety-critical systems (this includes establishing acknowledged coverage criteria and specification-based test oracles);
- Use of new notations and new kinds of modeling formalisms, e.g., property patterns, along with the elaboration of approaches based on usual programming languages;
- Integration of model-based testing into continuous development processes and environments (e.g., for software product lines).

This years MBT workshop features Alexandre Petrenko from the Computer Research Institute of Montreal, Canada, as an invited speaker. His talk and paper reflects on the use of nondeterminism in test models and the derivation of test cases from nondeterministic models. Furthermore, the accepted contributions, selected carefully by the program committee, show the above research trends. Christian Colombo, Mark Micallef and Mark Scerri report on testing of web services using a combination of tools and technics (Selenium, Cucumber and QuickCheck); in particular, they provide a natural language interface to a test generation tool. Harsh Beohar and Mohammadreza Mousavi address the conformance testing problem for software product lines; they introduce the idea of a spine which prunes the testing of a product derived from an already tested product. Adenilso Simao and Alexandre Petrenko generate finite complete test suites for predefined fault domains in I/O transition systems with the icoco conformance relation. Arjan van der Meer, Rachid Kherrazi and Marc Hamilton describe the combination of a model driven design tool (ASD) with an MBT tool (Spec Explorer), thereby combining formal analysis methods and model-based testing techniques. Kalou Cabrera Castillos, Frederic Dadeau and Jacques Julliand propose temporal property patterns over UML models for test case generation and show how to apply coverage criteria and mutations for robustness testing.

We would like to thank the program committee members and all reviewers for their work in evaluating the submissions. We also thank the ETAPS 2014 organizers for their assistance in the preparation of the workshop and the editors of EPTCS for help in publishing these proceedings.

Program committee

- Bernhard Aichernig (Graz University of Technology, Austria)
- Jonathan Bowen (Birmingham City University, Worshipful Company of Information Technologists, UK)
- Mirko Conrad (samoconsult GmbH, TU Munchen, Humboldt University of Berlin, Germany)
- John Derrick (University of Sheffield, UK)
- Bernd Finkbeiner (Universitat des Saarlandes, Germany)
- Lars Frantzen (Radboud University Nijmegen, Netherlands)
- Ziyad Hanna (Jasper Design Automation, USA)
- Philipp Helle (EADS, Germany)
- Mika Katara (Tampere University of Technology, Finland)
- Alexander S. Kossatchev (ISP RAS, Russia)
- Victor Kulaimin (ISP RAS, Russia)
- Bruno Legeard (Smartesting, France)
- Bruno Marre (CEA LIST, France)
- Laurent Mounier (VERIMAG, France)
- Nickolay Pakulin (ISP RAS, Russia)
- Jan Peleska (University of Bremen, Germany)
- Alexander K. Petrenko (ISP RAS, Russia)
- Alexandre Petrenko (Computer Research Institute of Montreal, Canada)
- Fabien Peureux (University of Franche-Comte, France)
- Holger Schlingloff (Fraunhofer FIRST, Germany)
- Julien Schmaltz (Open University of The Netherlands, Netherlands)
- Nikolai Tillmann (Microsoft Research, USA)
- Stephan Weißleder (Fraunhofer FOKUS, Germany)
- Nina Yevtushenko (Tomsk State University, Russia)

Additional reviewers

- Ylies Falcone (VERIMAG, France)
- Maxim Gromov (Tomsk State University, Russia)
- Wen-Ling Huang (University of Bremen, Germany)
- Omer Landry Nguena Timo (Computer Research Institute of Montreal, Canada)
- Uwe Schulze (University of Bremen, Germany)
- Christian Schwarzl (Virtual Vehicle, Graz, Austria)
Using Formal Specifications to Support Model Based Testing
ASDSpec: A Tool Combining the Best of Two Techniques

A.P. van der Meer  R. Kherrazi  M. Hamilton
Nspyre  Nspyre  Nspyre
arjan.van.der.meer@nspyre.nl  rachid.kherrazi@nspyre.nl  marc.hamilton@nspyre.nl

Formal methods and testing are two important approaches that assist in the development of high quality software. For long time these approaches have been seen as competitors and there was very little interaction between the two communities. In recent years a new consensus has developed in which they are seen as more complementary. In this report we present an approach based on the ASD(Analytical Software Design) suite by Verum and the Microsoft Spec Explorer Model Based Testing(MBT) tool. ASD is a model-based design approach that can produce verified software components that can be combined into complete systems. However, ASD cannot verify existing components, nor complex component interactions involving data transfers. We have developed a tool that allows us to convert ASD models to Spec Explorer, allowing us to do more complete verification of software systems using dynamic testing at little additional cost and effort. We demonstrate this by applying our approach to an industrial-size case study.

Keywords: Model Based Testing, Formal Verification, Dynamic Testing, Static Verification, EMF, ASD, Spec Explorer

1 Introduction

As with any kind of construction, reliability is of utmost importance in software engineering. To achieve this, Verum [3] has created Analytical Software Design (ASD), a component based design method. By using this method, software can be constructed that is guaranteed to implement specified interfaces and protocols and is deadlock-free. It is based on the decomposition of a system in different components modeled by state machines. These models can then be used to generate formally verified code that can be integrated to construct complete and correct industrial scale systems. However, in practice, many software systems require the use of third party and/or legacy components that have not been verified in addition to generated components. The ASD method cannot guarantee correctness for these external components nor for the integrated system, because it only performs static verification on a per-component bases and no dynamic testing nor testing of composed systems. To remedy this, we propose to use Model-Based Testing (MBT) to verify if the external components implement the behavior of the interfaces defined in the relevant ASD models, and if the complete system, including the external components, fulfills all requirements. We implemented a prototype of this approach that transforms ASD interface models to Microsoft Spec Explorer[14] models. Spec Explorer allows us to generate automated test cases that explore the expected interface behavior of any component, and enables us to find violations of the requirements with a minimum of manual effort. Furthermore, in addition to the verification ASD provides, Spec Explorer can use data testing to validate that the system implements desired behavior by linking inputs to observable events. Finally, Spec Explorer allows us to use model composition to test complete system, while also supporting data abstraction and combination [16] to reduce the state space to acceptable levels. In contrast, the facilities in ASD for handling data are very limited. By reusing
Using Formal Specifications to Support Model Based Testing

ASD interface models in Spec Explorer, this novel method gives us the full usage of all benefits of this powerful MBT tool (e.g. data abstraction to reduce state space and overcome state explosion and models decomposition) without facing issues related to creation of complex test models. In this paper, we will first introduce ASD and Spec Explorer in more detail in Section 2. We will then discuss the fundamental similarities that we use in our tool in Section 3. In Section 4 we describe some technical details of the implementation. In Section 5, we describe a study into the effectiveness of our tool and present some empirical results. Section 6 we discuss some related work, and finally conclusions in Section 7.

2 Preliminaries

2.1 Analytical Software Design (ASD)

(a) ASD specifications consist of design and interface models
(b) ASD guarantees correctness of generated code using static verification
(c) Spec Explorer verifies completeness of the system using dynamic testing

The ASD technology is developed by Verum with the primary aim of supporting the development of complex embedded software and increasing its quality. In the ASD approach, software systems are envisioned as a collection of components that communicate via interfaces. Figure 1a shows the overall structure of an ASD specification. To develop a new piece of software using ASD, we first have to specify one or more desired interfaces as interface models. As Figure 1a illustrates, these interfaces can involve hardware drivers (HW), user interfaces (UI) and other external components (EX). In addition to defining the operations, referred to as events, that the interface supports and the responses it can give, an interface model contains a state machine that defines the protocol that has to be used to access the interface. One of the main features of the ASD tool set is that these protocols are strictly enforced, so an ASD-generated software component can never violate them. Once the interface models have been defined, design models can be created that describe how to implement the interfaces. This is also done using state machines. An example of an ASD state machine is shown in Figure 1, with a visualization of the state machine on the left and a fragment of the definition created by the designer on the right. As shown, a state machine is constructed using a table layout, where for each state and for each possible event is defined how the component should respond. If the event is expected, the response can consist of changing the state of the component and sending responses to the calling component. Otherwise, the event is declared illegal, and if it occurs in the given state, this results in a failed verification. Overall, as shown in Figure 1b, this means ASD is focused on the core part of the system, while external components or hardware are not verified.
Once the interface models are complete, the next step is to create a design model that combines all interfaces. Like an interface model, a design model contains a state machine, but in this case it implements the behavior of the desired software component. To do this, a design model can refer to other interfaces, using them to provide some required functionality. When the design model is complete, ASD will verify that any used interfaces are always invoked according to their specifications. If the implementations of the interfaces are also generated with ASD, the resulting system is guaranteed to be correct with respect. However, if the implementation is actually third-party or legacy code, the component can be incorrect and invoke invalid events or respond incorrectly. In such cases, the generated system does not know how to respond and stops functioning. Verum suggests remedying this by introducing a so-called Armour layer between ASD systems and external components that filters any undesired communication, but this still means correct functioning of the system cannot be guaranteed. This is a fundamental limitation of the static testing concept used by ASD: anything not described directly by a model cannot be verified. In the case of larger systems with many components, ASD is also limited in its ability to verify complex interactions, because the number of states grows beyond the static testing it can do.

2.2 Spec Explorer

Spec Explorer is an extension to Microsoft Visual Studio intended to provide support for MBT. To use Spec Explorer, we first have to define a model that describes the expected behavior of the system to be tested. This model consists of one or more C\textsuperscript{\#} classes enhanced with modeling annotations. The model is used to compactly define the possible behavior of the system. Once the model is complete, we can apply testing strategies to generate test cases, which can be executed directly to see if the implemented system meets our expectations. In each model class, methods can describe behavior the system under test should implement. As any ordinary C\textsuperscript{\#} method, they can update variables, invoke other methods and return values. By computing the effects of each method, Spec Explorer can construct a state space containing all required behaviors. Each path in this state space represents a possible test, a sequence of steps that the system under test should be able to follow. Spec Explorer offers a range of strategies to select a representative sample of paths, based on for example data coverage. As shown in Figure 1c, because events are invoked on the system as a whole, this means not only the generated code but also external and even hardware components are involved in the tests.

How the system under test should execute the steps is described by the model annotations. Using the TypeBinding annotation, we relate each model class to an implementation class. Individual methods are linked to their counterparts using a Rule annotation. An example of a model is shown in Figure 2, which shows one method of a model class together with part of the state machine defined in the model. For readability purposes, we have removed states relating only to initialization and finalization details, and added labels indicating the correspondence between Spec Explorer states and ASD states as shown in Figure 1. The system under test is in this case actually an implementation of the alarm system also described in Section 2.1. This method describes how the alarm should react when it receives a triggered event. From the declaration of the method, we can see that it returns no value and has no parameters. In the body, we can see a switch statement that selects appropriate behavior based on the current state of the model, stored in the \texttt{AlarmSystemstatevar} variable. If the alarm is in the activated state, the first case of the switch will be used. This case specifies that the state of the model should be updated to triggered, and that the \texttt{IAlarmSystem.NI.Triggered} method of the \texttt{IAlarmSystem.NIimpl} variable should be invoked. In all other cases, the method cannot be used. This is indicated by the \texttt{Condition.isTrue(false)} construct, which indicates to Spec Explorer that the method call is not valid and should not be used in tests. If the method would be called in the implementation during testing, this would result in an
error and a failed test. In the state machine visualization, this method corresponds to the edge labeled \textit{triggered}. In addition to the model, Spec Explorer uses a script file in the CordScript language to define what tests should be executed. Figure 3 shows part of a CordScript file corresponding to the model shown in Figure 2, implementing a basic testing strategy. A CordScript file consists of two main kinds of elements: configurations and machines. Configurations, shown in the top part of Figure 3, define the basic parameters of the test, including the actions we are interested in and global properties like what test engine to use and how long tests can be. Machines, shown in the bottom part of Figure 3, are used to select which tests we want to execute. In most cases, we will want tests that cover all behaviors of the system, either as one large test or a number of smaller tests. If, for example, we want to test a specific action in the system, we can use a machine to select only those tests containing that particular action. In CordScript, we can also define model compositions to test complex systems and data abstraction and combination to reduce state spaces and test sizes. In the figure, we show part of a basic configuration, consisting of a number of switches that control the test generation process and two machines, one that defines the state space of the model, \textit{AlarmSystemProgram}, and one that defines the test strategy that we want to apply, \textit{AlarmSystemTestCases}. The end result is a number of state machines similar to the one shown on the left in Figure 3. The state machine represents the test case as a series of abstract steps, which can easily be translated to concrete steps which can be used by the chosen testing framework.

\begin{table}
\centering
\begin{tabular}{|c|c|c|c|c|c|c|}
\hline
\textbf{Interface} & \textbf{Event} & \textbf{Guard} & \textbf{Actions} & \textbf{State Variable Updates} & \textbf{Target State} \\
\hline
\multicolumn{6}{|c|}{
\begin{tabular}{l}
\textit{deactivated (initial state)}
\end{tabular}} \\
\hline
\multicolumn{6}{|c|}{
\begin{tabular}{l}
\textit{AlarmSystem SwitchOn} \\
\textit{AlarmSystem SwitchOn+} \\
\textit{AlarmSystem SwitchOff} \\
\textit{AlarmSystem SwitchOn+} \\
\textit{AlarmSystem SwitchOffHandled} \\
\textit{AlarmSystem INT [triggered]} \\
\end{tabular}} \\
\hline
\textit{StateInvariant} & - & & \textit{AlarmSystem.OK} & & \textit{activated} \\
\textit{AlarmSystem SwitchOn} & & & \textit{AlarmSystem.Failed} & & \textit{deactivated} \\
\textit{AlarmSystem SwitchOn+} & & & \textit{Legal} & & - \\
\textit{AlarmSystem SwitchOff} & & & \textit{AlarmSystem.VoidReply} & & \textit{deactivating} \\
\textit{AlarmSystem SwitchOffHandled} & & & \textit{Disabled} & & - \\
\textit{AlarmSystem INT [triggered]} & & & \textit{AlarmSystem, NilTriggered} & & \textit{triggered} \\
\hline
\end{tabular}
\end{table}

\begin{figure}
\centering
\includegraphics[width=\textwidth]{figure1.png}
\caption{Fragment of an ASD state machine definition and corresponding state diagram}
\end{figure}

\section{2.3 Conclusions}

In this section, we have described ASD and Spec Explorer. By explaining the strengths and weaknesses of both approaches, we want to establish what advantages can be gained by combining the two. In Figure 4, we show the fundamental idea in schematic form: we have a system that has to communicate with hardware(HW), other external components(EX) and a user interface(UI). During ASD verification, only the implementation of the new system is verified, while correctness of the other comments cannot be addressed. By using ASDSpec to generate Spec Explorer model based testing models, we can extend verification coverage from only the central components to the entire system. Additionally, we want to use...
Figure 2: Fragment of a generated Spec Explorer model and part of the corresponding state diagram

[Rule{Action = "this.triggered()", ModeTransition = "start4->start4"}]
public void IAlarmSystemINF_triggered()
{
    switch (this.AlarmSystemstatevar)
    {
        case AlarmSystemstates.activated:
        {
            this.AlarmSystemstatevar = AlarmSystemstates.triggered;
            this.IAlarmSystem_Nimpl.IAlarmSystem_NI_triggered();
            break;
        }
        default:
        {
            Condition.IsTrue(false);
            throw new InvalidOperationException();
        }
    }
}

Figure 3: Fragment of a CordScript script and part of generated test suite

switch StepBound = 128;
switch TestEnabled = false;
switch PathDepthBound = 128;
switch TestClassBase = "cs";
switch GeneratedTestPath = ".\AlarmSystem85.TestSuite";
switch GeneratedTestNamespace = "AlarmSystem85.ThreadTestSuite";

machine AlarmSystemProgram():Main
{
    construct model program from Main
    where scope = "AlarmSystemNameSpace"
}

machine AlarmSystemTestCases():Main where TestEnabled = true
{
    construct test cases
    where strategy = "shorttests" for(AlarmSystemProgram)
}
the facilities Spec Explorer offers in composition and abstraction to test larger, more complex systems. In Section 3, we will discuss the conceptual relations that make this possible in greater detail.

![Figure 4: ASDSpec converts ASD Interface Models to Spec Explorer Models and Cord Scripts](image)

3 Implementation Concept

In order to generate tests for a system using Spec Explorer, we need a model that describes the desired behavior and a script file that defines the kind of tests to be done. While creating these typically requires considerably little effort compared to that required to build the actual system, it is still a non-trivial process that can introduce its own errors. In the case we are testing ASD generated software, or software that has to be used in concert with ASD generated software, we already have a model that describes the required behavior. This suggests that if we can reuse this model, we can make testing with Spec Explorer easier and cheaper.

Because we cannot use ASD interface models in Spec explorer directly, this means we have to create a Spec Explorer model based on a given ASD model in a way that does not affect the semantics. In order to establish whether this is even feasible, we first have to discover what concepts ASD and Spec Explorer models share, and how they can be related. From a global perspective, we observe that both ASD and Spec Explorer are fundamentally state machine-based technologies. This means a system at all times has a well-defined state, which can change in response to triggers from the outside world. In particular, this implies that we can consider a Spec Explorer model compliant to an ASD model if the state machines involved are sufficiently equivalent in behavior. In particular, we want to make sure that the tests that are constructed by Spec Explorer can potentially contain all possible sequences of event triggers that are legal in the ASD state machine, to ensure we can reach any level of coverage we desire. On the other hand, illegal events should never occur in tests, because the behavior of the component is undefined in such cases, which means that the test can never be failed or passed.

Looking more closely, we see that in ASD, the state machines are implemented directly in the table-based language described earlier. Unfortunately, ASD is based on proprietary technology, and we do not have access to any formal definitions of the table language semantics. However, as mentioned before, the ASD tooling provides code generation support that generates executable code based on the state machines in several different languages. Based on the claims made by Verum, we can assume that the generated code for all languages are accurate representations of the intended state machine semantics. In this case, we look at the generated C\(^2\) code, because that is the language used by Spec Explorer. We see that all ASD interface are represented by C\(^2\) interfaces, and each ASD event is implemented as a method, declared in the appropriate interface and implemented in a C\(^2\) class. This means that events are triggered by calling the appropriate method, which is then executed when and how the C\(^2\) semantics dictate.
In Spec Explorer, the C\(^\#\) code of the model is used to define the potential behavior of a system. In particular, a Spec Explorer model contains one or more classes with methods, and each method describes an event that the system can respond to. By analyzing the effects of the methods on the state of the system, Spec Explorer can identify states and the transitions between them. Thus, to reconstruct a known state machine in a Spec Explorer model, we have to create one or more classes that together implement all possible events as methods, in such a way that the resulting state space matches the one in the ASD model. The former simply requires we create a method for every event. We achieve the latter by constructing the model based on an explicit state machine pattern, thus ensuring that all states are explicitly present in the model.

In order to execute actual test runs based on generated test cases, Spec Explorer requires direct links between model events and the corresponding methods in the system under test. Because this relation is similar to the connection between ASD model events and the methods that implement them, we can use our knowledge of the ASD code generation conventions to create these links. To make this more concrete, in Figure 5, we show a visualization of the structure of an ASD interface model on the left, and a visualization of corresponding C\(^\#\) code on the right. In the ASD model, we can see that the events supported by the interface model are defined in separate Application Interfaces. Separately, we can define a number of States, that for each event must describe how the system should respond to it. If the event is valid in the current state, this response consists of a transfer to a new state and possibly some actions. If the event is invalid, it is declared illegal, and the system is assumed to stop if it occurs in this state. On the C\(^\#\) side, we see that the interface model is implemented by several C\(^\#\) classes. The main class of the system is the Interface Model Component class. This class can be used to initialize the system, keeps track of the current state and provides access to implemented interfaces. Like in the ASD model, each interface contains a number of events, implement as methods. In order to trigger an event, we simply call the corresponding method. The implementation of the event will then use the current state recorded in the Interface Model Component to determine the correct behavior.

![Figure 5: Transformation Concept](image)

### 4 Implementation Approach

In Section 3, we have established that in order to create a Spec Explorer model based on an ASD model, we have to create a number of C\(^\#\) classes and methods that implement the corresponding state machine. To do this automatically, we choose to use a model transformation in QVTO [10], an Eclipse Modeling Framework (EMF) implementation of QVT Operational [12]. Because QVTO is based on EMF [8, 13], this means we have to translate ASD models into EMF form first. For this, we used an existing tool...
based on the XML schema provided by Verum, describing the structure of ASD models. This tool was developed by Nspyre as part of an earlier project. In the same project, a transformation was developed that abstracts from format-specific features of ASD models to a generic, more abstract representation. We use this representation as a basis for further processing. The next step is to generate the C♯ model file and the CordScript script file. This is implemented as two QVTO transformations. The first one generates an EMF C♯ model based on a C♯ meta-model created by the MoDisco [9, 2] project. The second generates an EMF CordScript model based on a CordScript meta-model of our own design. Because these are both in EMF format, we then have to use templates, in our case based on the Acceleo [7] template engine, to create the textual representations that can be used by Spec Explorer. These templates are generic, in the sense that they can be used for all EMF C♯ and CordScript models that use our metamodels.

5 Case Study

In order to demonstrate the effectiveness of our approach, we first created a prototype, applied it to a simple model, and tested its effectiveness using bug seeding. In a next step, we looked at a larger, more complex case study. The case we choose was developed earlier by Nspyre and is based on a container terminal that could be both simulated and implemented as a demonstration model. Because this system uses external components with legacy code, it serves as a good example of the advantages of model-based testing as complement to ASD verification. In the initial project, the required Spec Explorer test models were constructed by hand, at considerable effort, as is indicated in Table 1. This table is discussed in greater detail in Section 5.2. Using ASDSpec , we aim to reduce this cost.

5.1 Case description

An overview of the container terminal in question is shown in as a visualization in Figure 6, and in schematic form in Figure 7. As shown in the pictures, the terminal has three cranes, two of which, situated on the right, are primarily used for loading and unloading containers from the vessel(s), and one, situated on the left, that is used to move the cargo from or to other forms of transport. Goods are transported between cranes by an automated truck that is also part of the system. The main software component in the case is the controller, that has to move container to or from the right cranes in the right order. The system is judged to be working correctly when containers are moved to the right place safely and efficiently. For example, when moving a container to the truck, a crane should not release it until the vehicle is in position to support it, otherwise the container would fall to the ground.

Figure 6: Container Terminal Global Structure
5.2 Results

Based on the case study, we have drawn several conclusions on the approaches discussed in this paper, which are summarized in Table 1.

**Approach** General description of the verification approach

**Technique** ASD uses model checking to verify design models before code generation. In contrast, MBT creates tests based on manually created test models, that verify actual implementations by executing sequences of commands. ASDSpec constructs basic test models based on ASD models, which can be used directly to create compliance tests for components or extended to describe requirements not covered by ASD verification, such as those that are data-testing related [16].

**Modeling** ASD uses two kinds of models, interface models and design models. Interface models define behavior that can be provided by a component, and design models

**Effort** Because ASD design models need to cover every aspect of the implemented components, a significant effort is required to complete them. In return, ASD can save effort in the project as a whole through early design verification and code generation. In contrast, test models only need to cover the details needed for constructing suitable tests, so they are cheaper to construct, but they are useful only for testing. ASDSpec automatically generates test models based on existing modes, minimizing effort needed specifically for testing by reusing work that has been done earlier.

**Cost** In the original case study, the cost of modeling and constructing the MBT test cases and the test environment was estimated at 28 hours. Because we reused some results from that work, we cannot directly compare the time spend on the ASDSpec models with this figure, but as a conservative estimate, we computed a cost of 16 hours to reconstruct the test environment and the test models from scratch. This estimate includes overhead costs based on the original case study, which we expect could actually be reduced using ASDSpec, further reducing the cost of test creation.

**Tool support** As can be seen in Sections 2.1 and 2.2, ASD models are constructed using a table-based method, and Spec Explorer are defined by annotated C2 code. Because the ASD language is simpler and more structured, it is easier to define models than in Spec Explorer, at a cost in flexibility. Additionally, the ASD tooling automatically checks model completeness and consistency, which
means modeling errors can be detected at an earlier stage than in Spec Explorer. ASDSpec generates test models, which means the only complexity lies in any test customizations that are added later.

**Model composition** In software engineering, decomposition is a common method to reduce the complexity of systems. Some system-wide properties, however, can only be verified by examining all the parts of a system together. Purely component-based approaches, like ASD, are limited in their ability to handle these properties, because only interface models can be explicitly combined in design models, design models cannot be composed for verification purposes. In contrast, MBT test models can be combined to create test cases for entire systems, allowing specific test cases to be created when desired. Because ASDSpec is based on Spec Explorer, all model composition techniques available in that tool can be used with and added to our generated basic models.

**Number of test cases generated** In order to give an indication of the time needed to execute the verification, we look at the number of test cases used by each approach. Because ASD uses static verification only, there are no actual test cases involved. Instead, the time needed for verification depends primarily on the size of the state space, which is related only indirectly to the number of test cases. MBT does use test cases, and for our case study Spec Explorer generated 89 tests, based on combining the default shorttests and longtests test generation strategies and a manually created Spec Explorer model. If we apply the same strategies to the basic test models generated by ASDspec, we get 93 tests. While this number is likely to increase when modifications are made for specific requirements or model composition, it is an indication that the generated models initially possess a similar level of detail as the manually constructed Spec explorer models.

5.3 **Case Study Conclusions**

Based on our experiences in the case study, we conclude that the main advantage of the ASD approach lies in the combination of code generation and design verification, which allows validated components to be constructed at little cost over a direct, unvalidated implementation. However, the approach cannot be applied to (fully) verify existing components or systems containing them. In contrast, MBT requires at significant effort purely for verification, but can be applied to any system. Finally, ASDSpec also applies to both generated and external components, at greatly reduced costs, but the generated test models are based purely on the interface models, not on the specific properties of the system, limiting the flexibility of the generated tests. The automated approach does guarantee that the complete interface specification is represented in the model, and can thus be covered during tests.

6 **Related Work**

Software verification through both static and dynamic testing is a wide area of research, and we will not cover all of it here. Instead, we focus on the tools used in this paper: ASD::Suite and Spec Explorer. Starting with the first, there have been several papers on ASD and its use in industrial settings, for example [1, 11]. More interesting to us is [5], where ASD is combined with the model checking tool Uppaal to provide more complete verification, like we do here with Spec Explorer. In contrast with our approach, both ASD and Uppaal are based on static verification. The authors argue that Uppaal can handle more generic properties than ASD, in particular in the timing domain. This means the main benefit of combining the two tools lies in more detailed static verification of modeled systems, while our approach attempts to widen the scope of verification. Another contrast with our approach is that the
<table>
<thead>
<tr>
<th>Metrics</th>
<th>ASD</th>
<th>MBT</th>
<th>ASDspec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Approach</td>
<td>Generate deadlock-free code of components</td>
<td>Generate and execute test suite from behavior model</td>
<td>Generate test model/test suite from ASD interface model</td>
</tr>
<tr>
<td>Technique</td>
<td>ASD model checking</td>
<td>Model based testing</td>
<td>Static testing</td>
</tr>
<tr>
<td></td>
<td>Mathematical proof</td>
<td>Verification</td>
<td>Dynamic testing</td>
</tr>
<tr>
<td></td>
<td>Static testing</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Modeling</td>
<td>Create interface model + design model</td>
<td>Create test model</td>
<td>Generate test model form ASD interface model / generate from generated test model</td>
</tr>
<tr>
<td></td>
<td>Based on model decomposition</td>
<td>Supports model composition</td>
<td>Combines decomposition and composition</td>
</tr>
<tr>
<td>Effort</td>
<td>High</td>
<td>Medium</td>
<td>Low</td>
</tr>
<tr>
<td></td>
<td>Manually constructed models describe complete system</td>
<td>Manually constructed models describe relevant interfaces</td>
<td>Generated models describe relevant interfaces</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>28</td>
<td>16</td>
</tr>
<tr>
<td>Tool support</td>
<td>Medium</td>
<td>Low</td>
<td>High</td>
</tr>
<tr>
<td></td>
<td>Specialized table-based modeling language with strong tool support</td>
<td>Combination of two languages, need connection with system to be tested</td>
<td>Generated models, Cord-Script knowledge needed for test definition</td>
</tr>
<tr>
<td>Model composition</td>
<td>Interface model only</td>
<td>All test models</td>
<td>All test models</td>
</tr>
<tr>
<td></td>
<td>All test models</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of test cases generated</td>
<td>-</td>
<td>89</td>
<td>93</td>
</tr>
</tbody>
</table>

Table 1: Key Case Study Results
transformation from ASD to Uppaal is done by hand, while our approach is automatic. The authors do mention automation as a possible future extension of their work, because the manual procedure used is not very complex. Overall, our approach has the advantage that we use dynamic as well as static testing, allowing us to verify components that are not completely modeled, but for which implementations and interface models are available. In the same way, there have been several papers published on Spec Explorer. Most of these, like [4] and [15] focus on specific kinds of systems or testing strategies, while we focus on test model creation. To our knowledge, combining Spec Explorer with other tools is a novel approach that has not been described before. We believe that the ability to apply the functionality Spec Explorer provides, for example in the area of data combinations and model composition, make it worthwhile to extend the kind of projects it can be used for by creating connections with other tools.

7 Conclusions

In this paper, we have described a novel approach to verification that is based on combining model checking and model-based testing. To compare this approach with pure model checking and pure model based testing, we applied it to an existing case study of a container terminal. For model checking we use the ASD::suite tools both to perform static testing and code generation, doing no verification after the code has been generated. This approach has the lowest development cost of the three, while still offering some guarantees that the system meets all requirement, but it relies heavily on the assumption that external code implements interfaces correctly. The second approach, model-based testing, is implemented using the Microsoft Spec Explorer tool. In model-based testing, test models are used to generate test cases to verify implemented components and systems. While the extra modeling effort required increases the cost of this approach compared to the first one, components that where previously not verified can now be tested both in isolation and the context of an entire composed system. The third and new approach uses MBT based on models generated by ASDSpec based on ASD models. In the ASDSpec approach, we attempt to achieve all advantages of MBT without the extra model creation costs, by generating test models based on ASD interface definitions. In the case study, we confirmed that a significant reduction in cost of the testing process can be achieved. While the ASDSpec approach is less flexible than direct MBT, we feel it still provides significant verification for a variety of systems by extending ASD model checking with test results. As a next step, we intend to investigate combining advanced features of Spec Explorer with our tool, to further extend the verification possible, and to develop connections with other MBT frameworks, to access their unique features.

8 References


Verifying Web Applications: From Business Level Specifications to Automated Model-Based Testing

Christian Colombo  
Mark Micallef  
Mark Scerri

PEST Research Lab  
Department of Computer Science  
University of Malta, Malta

{christian.colombo | mark.micallef | msce0013}@um.edu.mt

One of reasons preventing a wider uptake of model-based testing in the industry is the difficulty which is encountered by developers when trying to think in terms of properties rather than linear specifications. A disparity has traditionally been perceived between the language spoken by customers who specify the system and the language required to construct models of that system. The dynamic nature of the specifications for commercial systems further aggravates this problem in that models would need to be rechecked after every specification change. In this paper, we propose an approach for converting specifications written in the commonly-used quasi-natural language Gherkin into models for use with a model-based testing tool. We have instantiated this approach using QuickCheck and demonstrate its applicability via a case study on the eHealth system, the national health portal for Maltese residents.

1 Introduction

In typical commercial contexts, modern software engineering is characterised by a strong customer focus and, as a result, fluctuating requirements. The industry has developed ways to adapt to this, most notably through Agile development processes [5] which have gained substantial popularity over the past decade. These processes start off by gathering requirements from the customer in the form of case scenarios and subsequently development is built around these scenarios culminating in their implementation and demonstration to the customer. This process, better known as behaviour-driven development [3], reorients the development process so as to ground all work around business level customer specifications. Once the customer scenarios are implemented, the process tends to take on an iterative form whereby the customer is again asked to review the specification and possibly introduce further scenarios. To keep the specification language as simple as possible for non-technical customers, scenarios are typically specified in terms of a quasi-natural language called Gherkin [8]. In order to automate test execution, clauses from the Gherkin specification are associated with executable code using a technology such as Cucumber [6]. This executable code effectively turns the specification into an automated acceptance test which is used to test that the system adheres to the customer’s specifications.

Whilst the automation achieved through the running of Gherkin specifications has significantly shortened the test execution process and made it repeatable, a significant limitation is that the scenarios are never executed in combination with one another: one would typically expect some classes of problems to arise through the interactions between a sequence of scenarios rather than within the context of individual scenarios. In this context, we present an approach to combine related scenarios into models which can be consumed by model-based testing tools. In turn, such tools can be used to explore the model, generating plausible sequences of scenarios that provide more thorough testing of the system for minimal extra effort. We show how this technique has been instantiated on the QuickCheck testing tool and applied to Malta’s national health portal web application. This paper makes three main contributions:
(i) A technique for automatically generating system models from existing test case scenarios written in the Gherkin language. These models can be consumed by model-based testing tools in order to generate test cases that extend normal testing and facilitate inter-scenario interaction. (ii) A proof-of-concept tool which applies the technique presented here to generate models for the QuickCheck test case generation tool. (iii) A case study on Malta’s national health portal which demonstrates that the technique can be applied to a real-world system.

In order to control scope at this stage of our research, we constrain our work to the context of web applications. Whilst this is a plausible constraint due to the increased proliferation of web applications, it does not preclude the technique from being modified and applied to other contexts. The rest of the paper is organised as follows: Section 2 provides the reader with the background required to understand the work presented in this paper whilst Sections 3 and 4 explain the proposed technique and provides an overview of the instantiation respectively. This is followed by Section 5 whereby we discuss a case study designed to demonstrate the applicability of the technique. Finally, Section 6 discusses related work from the literature and Section 7 features our conclusions and opportunities for future work.

2 Background

Gherkin is currently an industry standard language enabling non-technical customers to express requirements in terms of case scenarios. Ensuring that the developed system adheres to these specifications is crucial for customer acceptance of the resultant system. By associating executable code to statements within Gherkin, developers can use tools such as Cucumber to automatically execute test scenarios on the underlying system. However, to enable the automated manipulation of web applications through a browser, mimicking the user, one would need a tool such as Selenium which provides the necessary interface to access a web application programmatically. The rest of this section provides the necessary detail on these technologies so as to enable the reader to understand the work presented here.

2.1 Gherkin

Gherkin [8] is a business readable, domain-specific language which lets users describe what a software system should do without specifying how it should do it. The language forms an integral part of Behaviour Driven Development [3], a development process which aims to make specification a ubiquitous language throughout a system’s life cycle. Gherkin specifications take on a so-called Given-When-Then format as illustrated in Example 1 and the underlying concept is that a business user can specify what is expected from a system using a series of concrete scenarios. Given some precondition, When the user does something specific, Then some postcondition should materialise.

Example 1
1. Feature: Doctor’s landing page
2. Scenario: Navigate to lab results page
3. Given I am on the doctors landing page
4. When I click on laboratory results
5. Then I should go to the lab results page
6. Scenario: Navigate back to Doctor’s landing page
7. Given I am on the lab results page
8. When I click on the myHealth logo
9. Then I should go to the doctors landing page

In this example, the client has specified two requirement scenarios as part of a feature on an eHealth system: one dealing with the process of a doctor viewing the laboratory results and another one of a doctor navigating back to the landing page.
Developers then decompose each scenario into a number of development tasks and the work proceeds with all activity being a direct result of a particular customer-specified scenario. Furthermore, although the scenarios are written in a subset of natural language, technologies exist (e.g. Cucumber [6] and Specflow [14]) which enable developers to implement automated user acceptance tests for every step in a scenario. That is to say that (i) the user specification becomes executable as an automated test by the end of a development iteration; and (ii) any future scenarios which include a predefined step will automatically reuse test automation code, thus in effect building up a domain specific language for automated testing.

2.2 Cucumber

Cucumber [6] is a technology that enables developers to provide glue code which turns natural language specifications into executable automated tests. Although originally written in Ruby, the tool has been ported to a number of languages. It reads plain language text files called feature files (as in Example 1), extracts scenarios and executes each scenario against the system under test. This is achieved by means of so-called step definitions, a series of methods which are executed when a step in a feature file matches a particular pattern. For example for the statement When I click on the myHealth logo, the developer associates code which actually simulates a user clicking on the logo. Once all Gherkin statements have associated code which can be executed by Cucumber, the latter will iterate through each step, find a matching step definition and execute the method associated with it. Step definitions can implement any feature supported by the underlying language. In our case, we are interested in manipulating web browsers for automated web application testing; something which can be achieved by Selenium.

2.3 Selenium

Selenium [13] is a web browser automation framework which has become a de-facto industry standard when it comes to web test automation. It has been implemented in a number of popular programming languages and provides API-level access to various browsers. In essence, developers can write programs which launch a browser, simulate user behaviour within the browser and subsequently query the state of website elements to verify expected behaviour as illustrated in Example 2.

Example 2

```java
public void testGoogle() {
    // Launch browser and visit www.google.com
    WebDriver browser = new FireFoxDriver();
    browser.open("http://www.google.com");
    // Search for “Hello World”
    browser.findElement(By.id("q")).sendKeys("Hello World");
    browser.findElement(By.id("btnG")).click();
    // Check that the search engine behaved appropriately
    assertEqual("Hello World - Google Search", browser.getTitle());
}
```

The above code snippet features source code written in Java which carries out a test on the Google search engine using the Firefox browser. The WebDriver object in this example is provided by the Selenium framework.

Combining Cucumber with Selenium can result in truly powerful automated test suites which provide repeatable and consistent regression testing capabilities. We argue that this combination can be leveraged even further by bringing model-based testing into the mix at minimal cost by constructing models from existing Gherkin specifications.
3 Proposed Technique

Starting with a number of Gherkin scenarios, the proposed technique combines characteristics inherent in the Given-When-Then notation with those of web applications in order to make educated inferences and consequently construct a model of a specified system. In order to construct a model representation of web applications, the specification language should allow us to:

1. Define the states of the system
2. Specify the starting state
3. Specify the transitions available at each state, including any preconditions, the action which causes the system to change state, and any postconditions

We argue that the Given-When-Then structure of Gherkin lends itself to providing this information since a single scenario in Given-When-Then form can be interpreted to represent a single transition between two states. That is to say that the Given step indicates an origin state (and possibly preconditions), the When state specifies actions for a transition to occur, and finally the Then step indicates the target state (possibly with postconditions). However, in order to facilitate the automated interpretation of scenarios in this manner, we propose that users of the language adopt three conventions taking into account the navigation-driven nature of web applications. To illustrate the conventions, we will be using Example 3 as a running example.

Example 3
1. Feature: Patient management features
2. Scenario: Navigate to lab results page
3. Given I start on the "doctors landing page"
4. And I have pending lab results
5. When I click on laboratory results
6. Then I should go to the "lab results page"
7. And I should see my pending lab results
8. Scenario: Navigate back to Doctor’s landing page
9. Given I am on the "lab results page"
10. When I click on the myHealth logo
11. Then I should go to the "doctors landing page"

The above Gherkin code consists of two scenarios which are authored according to the conventions proposed here. More details on the pattern of these scenarios are given below when describing the conventions.

Corresponding to the three points outlined about, we propose the conventions below:

Convention 1: Make states explicit. In order to identify states within a test scenario, we constrain the language such that Given steps take the form of Given I am on the "<web page>". Similarly, Then steps take the form of Then I should be on the "<web page>". Recall that we consider web pages within an application to represent states within the application’s model. This convention, whereby Given and Then steps take a specific form which includes state names being surrounded by quotes, enables us to clearly identify states within the model. In Example 3, the states “doctors landing page” and “lab results page” can be extracted from lines 3, 6, 9 and 11. Upon closer inspection, one realises that line 3 takes on a slightly different form. This is outlined in the next convention.

Convention 2: Identify start states. In order to identify start states in a model, we propose that the convention on Given steps consist of two variations. The first is the Given I am on the "<web
Conventions of specifying web application test scenarios. The first example outlines "Given I start on the "<web page ">". This minor variation will identify the web page in question as a starting state in the system’s model.

Convention 3: Identify actions, preconditions and postconditions. Finally, we identify the When construct as the action which takes the user from one page to another while the And steps which form part of Given or Then parts of scenarios will be leveraged to identify preconditions and postconditions respectively. That is to say that this additional information can be used to infer conditions which should be satisfied before a transition can fire, as well as conditions which should be satisfied after a transition has fired. Consider the first scenario in Example 3. The action taking the user from the “doctors landing page” to the “lab results page” is identified by the When clause: “clicking on laboratory results”. Furthermore, in this case, the Given portion of the scenario has two lines. The first identifies the current state as the “doctors landing page” whilst the second sets up a precondition on the subsequent transaction. In order for “clicking on laboratory results” to be a valid action, first the condition that the doctor has pending lab results must be satisfied. Similarly the Then portion of the scenario identifies the “lab results page” as the target state of the scenario whilst line 7 states that the doctor should also see pending lab results within that page once the transition is complete.

If these conventions are adhered to, test scenarios can be interpreted as follows:

**Given:** The Given step is usually used to specify preconditions in a test scenario but in the context of model-based testing, it can be used to infer the current state of the system. In accordance to convention 2, a special adaptation of this step can also be used to specify the starting state of a model. Furthermore, any And steps immediately following the Given step will be treated as preconditions.

**When:** In normal use, the When step specifies actions for exercising the system under test in a scenario and retains this role in our technique such that the action specifies a transition from the current state to a target state.

**Then:** The Then step is usually utilised to specify a postcondition in a test scenario but in our technique we adapt this to specify the target state. Furthermore, any And steps immediately following the Then step will be treated as postconditions.

By processing multiple test scenarios and joining them at points where states from different scenarios have the same name, a model can now be constructed. This is demonstrated in Figure 1 whereby four generic test scenarios are combined to construct a single model. More formally, given a list of test scenarios written in Gherkin such that they adhere to the conventions discussed above, a model can be constructed using Algorithm 1. In short the algorithm processes the scenarios, identifies unique states and inserts the transitions including actions, preconditions and postconditions. Figure 2 shows a model constructed by applying this algorithm to Example 3.

In the next section, the approach described above is instantiated in the context of a particular model-based testing tool.

4 Instantiation

As a prerequisite to evaluating the approach, we developed an instantiation: namely stringing together three parts: (i) the algorithm translating the Gherkin scenarios into a model for a model-based testing
Data: scenarios : List of Test Scenarios

Result: model : A finite state machine modelling the system under test.

```
model = new Model();
while scenarios.hasMoreScenarios() do
  scenario = scenarios.nextScenario();
  os = scenario.extractOriginState(); // According to Convention 1
  ts = scenario.extractTargetState(); // According to Convention 1
  if (os != null ∧ ts != null) then
    model.addStateIfNew(os);
    model.checkIfStarting(os); // May be marked as start state if Convention 2
    preconditions = scenario.extractPreconditions(); // According to Convention 3
    postconditions = scenario.extractPostconditions(); // According to Convention 3
    actions = scenario.extractActions(); // All actions from the When steps
    if (actions != null) then
      transition = new Transition(preconditions, actions, postconditions);
      model.addTransition(os,ts,transition);
    else
      error(Unable to extract transition from test scenario);
  else
    error(Unable to extract states from test scenario);
  end
end
return model;
```

Algorithm 1: High level algorithm for generating models from business specifications.

Figure 1: A depiction of how models can be constructed from multiple test scenarios.
tool, (ii) the model-based testing tool which generates the test cases and detects test failure or success, and (iii) Selenium which enables the testing tool to interact directly with the web application.

While other model-based testing tools such as ModelJUnit\(^1\) are perfectly valid alternatives, as a model-based testing tool, we selected QuickCheck [4] for the fact that it provides test case shrinking functionality whereby upon detecting a failure, it searches for smaller test cases which also generate the fault to aid debugging. Although initially developed as a tool for the Haskell programming language, it has been implemented on a number of platforms and for this work, we utilise the Erlang implementation [2]. QuickCheck is primarily a random test data generation tool but it also supports model-based testing. Given a model and a set of pre and postconditions, the tool is able to automatically generate test cases which are used to verify that the conditions hold in the context of different sequences of actions. Figure 3 illustrates an example QuickCheck model which represents valid sequences of operations involving a stack (top elements in square brackets represent preconditions while the bottom elements represent postconditions). Elements can be pushed or popped from the stack: Popping an element from the stack is not allowed when the stack is empty, otherwise the size should be decremented by one. On the other hand, pushing an element should result in the size of the stack to increase by one.

A QuickCheck test case execution starts from the initial state and selects a transition whose precondition is enabled. The action upon the transition is actuated and the postcondition is checked. If a

\(^1\)http://www.cs.waikato.ac.nz/~marku/mbt/modeljunit/
postcondition check fails, then this signifies a failed test.

Once QuickCheck has been chosen to instantiate our approach, a tool was developed as illustrated in the architecture diagram in Figure 4. The system takes a set of test scenarios as input and proceeds to parse them and convert them into a QuickCheck model. This model is then fed to QuickCheck for test case generation and execution. During the test case execution process, Selenium is utilised for interacting with a web browser and exercising the web application. Finally, QuickCheck outputs a report containing results of the test run.

Having implemented an instantiation of the technique, we could explore its applicability through an industry case study. This is the subject of the following section.

5 Case Study

Our initial evaluation of the technique presented here is a case study. The objectives of this case study were threefold:

1. Demonstrate the feasibility of the technique on a non-trivial system
2. Compare the technique to its manual alternative
3. Note any relevant observations to help form future research directions

The case study was carried out on myHealth\(^2\), a web-based solution that has recently been developed for the government of Malta’s Ministry of Health to provide an improved and more efficient service to patients in the Maltese health care infrastructure. It consists of a web portal allowing access to myHealth data for registered citizens. A back-end portal is provided so as to allow monitoring and management to administrators. The case study was scoped to the front-end portal only, mainly because the back-end portal was purchased off the shelf and no test scenarios were available for it. The system is able to distinguish between doctors and citizens, providing different functionality for the two roles with Table 1 providing a list of high level functionality available to both types of users. When citizens gain access to

\(^2\)http://www.myhealth.gov.mt
their myHealth Record account, they are able to search for doctors listed in the Medical Council Register, request a specific doctor to be their trusted doctor and go on to view and manage all their personal health data.

The case study was carried out as depicted in Figure 5. Since the system’s test scripts were not documented in Gherkin format, they needed to be analysed and written in the required form. Once this was done, test automation code was created such that all the scenarios became executable automated tests. This enabled us to feed the test scenarios into our tool, create QuickCheck models and subsequently generate and execute tests based on the models. These steps are elaborated in the following subsections.

5.1 Constructing a model for the case study

When we were granted access, the myHealth Record system had already been through testing and been launched to the public. In this section we briefly outline the testing process used by the company that developed myHealth. The system went through two phases of testing: the first phase was a unit testing and integration phase which was mainly carried out by developers in the company. The second phase consisted of user acceptance testing and was carried out in conjunction with the customer in order to verify that the required functions had been implemented and were working correctly. The nature of the work presented in this paper is mainly concerned with system-level or user acceptance testing, so the second phase is of interest to us. This phase was driven by pre-documented test cases which outlined a series of procedures and expected outcomes (see Table 2 for a sample test case).

For the purposes of this case study, the test cases from the second phase of testing were secured and used as the basis to construct specifications in the Gherkin language. Each test case was manually analysed and systematically converted to its equivalent Given-When-Then format. Furthermore, step definitions were implemented such that each scenario became an executable automated test. Related test scenarios were then grouped together and models were generated accordingly using an implementation of Algorithm 1. Figure 6 shows one of the models generated during the exercise.

5.2 Results and Observations

With regards to exploring the feasibility of such an approach on a non-trivial system, the case study demonstrated that this is indeed the case. We were able to generate models directly from business level specifications and subsequently utilise those models to generate and execute test cases. The process took approximately seven days of work to complete. This was similar to the length of time taken by the company to carry out manual user acceptance testing of the product. However, if one looks closer at how the time was invested (see Table 3), in the case of manual testing, the company estimates that four days were spent designing test cases whilst three days were spent executing them. In the case of automated model-based testing, the test design aspect was practically free (due to the use of QuickCheck) whilst the actual execution only took one day. Whilst on the face of it, it seems that both approaches are equally time consuming, it is worth noting that if the company had adopted the use of Cucumber...
<table>
<thead>
<tr>
<th>Actions available to doctors</th>
<th>Actions available to citizens</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Login as a doctor</td>
<td>• Go to appointments page</td>
</tr>
<tr>
<td>• Go to appointments page</td>
<td>– View appointments</td>
</tr>
<tr>
<td>– Search for a patient’s appointments</td>
<td>• Go to case summaries page</td>
</tr>
<tr>
<td>– View a patient’s appointments</td>
<td>– Search for case summary</td>
</tr>
<tr>
<td>• Go to case summaries page</td>
<td>– View a case summary</td>
</tr>
<tr>
<td>– Search for a patient’s case summary</td>
<td>– Print case summary</td>
</tr>
<tr>
<td>– View a patient’s case summary</td>
<td>• Go to laboratory results page</td>
</tr>
<tr>
<td>– Print patient’s case summary</td>
<td>– Search for a lab result</td>
</tr>
<tr>
<td>• Go to laboratory results page</td>
<td>– View a lab result</td>
</tr>
<tr>
<td>– Search for a patient’s lab result</td>
<td>– Mark lab results as read or unread</td>
</tr>
<tr>
<td>– View a patient’s lab result</td>
<td>– Print lab results</td>
</tr>
<tr>
<td>– Release result to patient</td>
<td>• Go to medical image reports page</td>
</tr>
<tr>
<td>– Mark lab results as read or unread</td>
<td>– Search for a report</td>
</tr>
<tr>
<td>– Print patient’s lab results</td>
<td>– View a report</td>
</tr>
<tr>
<td>• Go to medical image reports page</td>
<td>– Mark medical image report as read or unread</td>
</tr>
<tr>
<td>– Search for a patient’s report</td>
<td>– Print medical image reports</td>
</tr>
<tr>
<td>– View a patient’s report</td>
<td>• Go to POYC entitlement page</td>
</tr>
<tr>
<td>– Release report to patient</td>
<td>– View a POYC entitlement</td>
</tr>
<tr>
<td>– Mark medical image reports as read or unread</td>
<td>– Print entitlement</td>
</tr>
<tr>
<td>– Print patient’s medical image report</td>
<td>• Go to search for doctors page</td>
</tr>
<tr>
<td>• Go to POYC entitlement page</td>
<td>– View all registered doctors</td>
</tr>
<tr>
<td>– Search for a patient</td>
<td>– Search for a doctor</td>
</tr>
<tr>
<td>– View a patient’s entitlement</td>
<td>– Send request to a doctor</td>
</tr>
<tr>
<td>• Go to search patients’ data page</td>
<td>• Go to manage doctors page</td>
</tr>
<tr>
<td>– Search for a patient</td>
<td>– Remove a doctor</td>
</tr>
<tr>
<td>– View a patient’s records</td>
<td>• Go to manage notifications page</td>
</tr>
<tr>
<td>• Go to manage patients page</td>
<td>– Subscribe to email notifications</td>
</tr>
<tr>
<td>– Search for a patient</td>
<td>– Subscribe to mobile notifications</td>
</tr>
<tr>
<td>– Remove a patient</td>
<td>• Go to manage notifications page</td>
</tr>
<tr>
<td>• Go to manage notifications page</td>
<td>– Subscribe to mobile notifications</td>
</tr>
</tbody>
</table>

Table 1: Functionality available to the doctors and citizens in the myHealth system.
Test Case Number: 4

Scope: Releasing the full Updated Report through the Patient Data menu

Procedure and Expected Outcome:

Doctor - Releasing the report

1. Login to the system as Doctor - e-ID 0240860M
2. In the Doctor Landing Page, open the Patient Data accordion
3. Navigate to the Medical Image Reports section in the accordion
4. Locate the previous report and Click on View Reports
5. Click Release All
6. Logout

Citizen - Testing that the updated report has been released

1. Login to the system as Citizen - e-ID 0240851M
2. In the Citizen Landing Page, open the myMedical Image Reports accordion. Note that the Updated box in the accordion shows 1
3. Locate a report and note that an Updated Reports icon is displayed
4. Click on View Reports and note that some of the reports were updated. Previous reports should be marked as Superseded and greyed out while the updated report should be marked as Final (Updated). The report should contain the updates.
5. Logout

Date Run: 01/03/2012
Run By: <Names of testers present>
Overall Test Outcome: Pass

Additional Commands and/or Screenshots:

Table 2: A sample test case from the user acceptance testing phase
in its normal operations (as is the case with an increasing number of companies), the time require for automated model-based testing would have been substantially reduced. Furthermore, automated model-based testing is efficiently repeatable and will deliver consistent results when compared to that delivered manually by human testers.

Two main observations were made throughout this process: Firstly, automated testing will not be able to replicate all tests which are carried out manually by a human tester. In fact, we were able to replicate 74% of manual tests due to test cases which either required human judgement (e.g. checking printouts) or were too complex to feasibly automate (e.g. temporal tests). The second observation was related to the shrinking feature of QuickCheck: while this particular feature is fast in console-style applications, we observed that shrinking took an inordinately long amount of time when testing web applications. This is mainly due to the expense required to repeatedly launch a browser and navigate through a series of web pages during the shrinking process.

It is worth noting that no new bugs were uncovered as a result of applying the technique. We think that this is due to the fact that by the time we ran the case study, the system had been thoroughly tested, deployed and in active use for quite some time.

![Model generated for myHealth's Doctor's Lab Results functionality](image)

**Table 3:** Comparison of the time required for manual testing and automated model-based testing.

<table>
<thead>
<tr>
<th></th>
<th>Manual Testing</th>
<th>Automated Model-based Testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generation of test cases</td>
<td>4 days</td>
<td>-</td>
</tr>
<tr>
<td>Creating and automation Gherkin Scenarios</td>
<td>-</td>
<td>6 days</td>
</tr>
<tr>
<td>Execution of test cases</td>
<td>3 days</td>
<td>1 day</td>
</tr>
<tr>
<td><strong>Total time</strong></td>
<td>7 days</td>
<td>7 days</td>
</tr>
</tbody>
</table>
5.3 Threats to Validity

The evaluation presented here is mostly concerned with demonstrating that the automated generation of models from natural language specifications and the subsequent generation and execution of tests is prima facie feasible. Consequently, the case study is limited in terms of depth and is subject to a number of external validity threats. We split these threats into two categories: (i) threats related to new challenges introduced by this technique and (ii) threats related to test engineering challenges which are manifested in most automation attempts. We argue that the first category of threats are highly plausible and should be investigated further whilst the second category of threats is diminished due to the fact that we are reusing existing automation code which should have already dealt with them.

The main threat introduced by the current approach is concerned with the requirement that conventions be followed when constructing natural language specifications. That is to say that preconditions, actions and post conditions are assumed to reside in specific elements within each scenario. Whilst the conventions are arguably reasonable, they do introduce some restrictions on what is meant to be a flexible language. This constitutes a threat to validity because unlike this case study whereby scenarios were created for the purpose of the evaluation, real-life scenarios will be written by different people who may or may not have followed the same conventions.

With regards to test engineering challenges, two representative issues come to mind. Firstly, one can also argue that the data used in this case study was not sufficiently complex and in reality, data dependencies may be highly complex. Whilst the concern is justified, we argue that the threat to validity here is limited, mostly due to the fact that the technique assumes that an automated testing infrastructure already exists. Such infrastructures usually contain sophisticated test data factories which support automated testing. These factories are typically called by ‘Given’ steps in order to create/fetch any required data as well as by ‘Then’ steps to tear down and residual test data that is no longer required. Similar arguments can be made for other classes of test engineering challenges. For example, some test could be sensitive to timing issues. However, since we are essentially reusing existing test automation code which is exposed to the same challenges, it can be safely assumed that test engineers would have solved these challenges within that code. If not, then the original set of automated test would not be functional in the first place. Whilst this should reduce validity concerns related to data dependencies, we do intend to observe this in more elaborate case studies as discussed in Section 7.

6 Related Work

The application of model based testing to web applications is not new and has been proposed on numerous occasions. Andrews et al. [1] proposed the use of finite state machines as an ideal model for this domain. Marchetto et al. [11] extended this approach to cater for more dynamic Ajax-based applications. Similarly, Wu and Offutt [15] propose a technique to model both static and dynamic aspects of web applications based on identifying elements of dynamically created web pages and combining them using sequence, selection, aggregation and regular expressions to create composite web pages. Ernits et al. [7] demonstrate how model based testing can be applied to web applications using the NUnit tool. NUnit allows developers to create models in C# and subsequently uses those models to generate test cases. Whilst the authors had success with the technique, they commented that a significant amount of time was spent learning the modelling formalism and building the test harness [7]. This contrasts to our approach which bypasses the need of manually programming the models as long as the business-level specifications adhere to the conventions highlighted in this paper.
With regards to automatic model generation, the literature mostly reveals work utilising reverse engineering approaches. Hajiabadi and Kahani [9] propose a technique for first modelling the structural aspects of a web application by observing client requests and server responses, and subsequently applying ontologies to generate test data. The authors claim that ontologies are required in order to make test case selection more effective. Our approach differs in that ours focus on the generation of sequences of actions rather than the data used. (This is left as future work in our case.) Ricca and Tonella [12] propose a UML representation of web applications and present a tool called ReWeb which reverse engineers web applications into this notation. They also present a tool named TestWeb which processes UML representations of web sites to automatically generate and execute test cases. While this approach is similar to ours in the use of a high-level specification notation, it differs in that it uses the implementation of the system itself to extract the model while we use the high-level business specification to this end.

The closest work to ours is that carried out by Jääskeläinen et al. [10] which similarly synthesises and merges test models from simple linear test cases. The main difference is that our propositions are not state-based but rather transition-based, resulting in simpler states. The implication is that our merging is fully automatable (with the possibility of having duplicate transitions) while in their case the merging has to be manually checked.

7 Conclusions and Future Work

In this paper we proposed an approach for converting business level specifications written using Given-When-Then natural language notation into models that can subsequently be used to generate and execute test cases. Whilst the literature contains multiple proposals for model-based testing of web applications, most techniques require the involvement of highly technical individuals who were capable of constructing models of their systems. Given the dynamic evolutionary nature of modern systems, maintaining models in the long run may not be desirable or even feasible. Our technique can work off natural language specifications which adhere to three simple conventions as discussed in Section 3 and if applied in a company where automated testing is already the norm, model creation and subsequent test case generation can be achieved at minimal cost.

With regards to future work, there are a number of paths which we would like to pursue:

- The first involves converting the technology to make use of different model-based testing tools on various platforms. This is desirable because the most expensive part of the approach is the automation of test scenarios in the language of the model-based testing tool, which in our case was Erlang. If models can be generated in the same language as that being used by developers in a particular company, then test automation code can simply be reused by models out of the box.

- The second path of research involves carrying out larger case studies both in terms of system size, as well as in terms of project duration. It would be ideal to find industry partners who have ongoing projects using Cucumber as a test automation tool and would like to utilise our technique on a long term basis. This would help us make more detailed observations and modify the technique accordingly.

- While our approach is a perfect fit for web applications due to the ease with which scenarios can be connected over common web pages, we are confident the approach can also be applied to other domains such as desktop GUI-based applications where the connecting element might be a tab or a window which is enabled at a particular point in time.
• On a more theoretic level, we wish to explore the possibility of modelling variations of data elements rather than limiting ourselves to sequences of actions. For example in the case study we could model different laboratory results, doctor attributes, patient conditions, etc., automatically to generate more varied test cases.

• Finally, it is rarely the case that scenarios run in a sequential fashion in a real life environment. Thus, it would also be interesting to explore models which allow modelling several users concurrently executing different scenarios.

References


Coverage Criteria for Model-Based Testing using Property Patterns

Kalou Cabrera Castillos
LAAS-CNRS
BP 54200
31031 Toulouse cedex 4, France
kcabrera@laas.fr

Frédéric Dadeau
FEMTO-ST Institute/INRIA CASSIS Project
16 route de Gray
25030 Besançon cedex, France
frederic.dadeau@femto-st.fr

Jacques Julliand

We present in this paper a model-based testing approach aiming at generating test cases from a UML/OCL model and a given test property. The property is expressed using a dedicated formalism based on patterns, and automatically translated into an automaton. We propose new automata coverage criteria that are tailored to the property automata we consider. These criteria are based on the coverage of a relevant subset of the transitions related to the original property, aiming at producing test cases that illustrate the dynamics of the system described in the property. In addition, we propose to produce test cases that can ensure the robustness of the system w.r.t. the property, by mutating the property automaton, in order to try to provoke events that would be forbidden by the property. This approach has been implemented into a large tool set and it has been experimented on realistic case studies, in the context of industrial research projects.

Keywords. Model-Based Testing, UML/OCL, property patterns, coverage criteria, automata mutation.

1 Introduction and Motivations

Model-Based Testing is one of the best ways to automate software testing. Even if the design of the model is a costly task, it is damped by the profits that can be made from it. Indeed, the model can be used to automatically compute the test cases, from simple input test data to complete sequences involving several steps. In addition, it is used to provide the oracle, namely the expected results of the test, thus deciding of the conformance between the system under test and its formal representation. Many model-based software testing tools exist, such as Smartesting’s CertifyIt [8], Conformiq Designer [17], or STG [9].

In practice, some of these tools work by automatically applying a structural test selection criterion on the model (such as the structural coverage of the OCL code [23] of the operations contained in an UML class diagram). Even if this approach is quite effective in practice, it suffers from its subjectivity and thus, specific behaviours of the system, which require a more intensive test effort, are not much targeted by this testing strategy. To overcome this problem, dynamic test selection criteria are introduced. They consist in scenario-based testing approaches that aim at exercising more extensively specific parts of the considered system.

In general, test scenarios are expressed in a dedicated language that can be either textual [5] or graphical [9], describing sequences of steps (usually operation calls) that can be performed, along with possible intermediate states reached during the unfolding of the scenario. Nevertheless, the design of the test scenarios remains a manual task that we aim to automate. During previous experiments in the use of scenarios, we have noticed that scenarios often originate from a manual interpretation of a given property that exercises the dynamics of the system [18]. Our goal is now to express such properties, in a simple formalism, that can be later exploited for testing purposes. To achieve that, Dwyer et al. introduced...
the notion of property patterns that makes it possible to express dynamic behaviours of the systems without employing complex temporal logics formalism [12]. Such properties are expressed with a scope, that delimits some fragments of the execution of the system, and a pattern that expresses occurrences, absences or chains of given events inside the considered scope.

The proposed process is depicted in Fig. 1. A test property $\varphi$ is described using a property pattern that holds on the UML/OCL model. This property is translated into an automaton $A_{\varphi}$, which captures the events that are described in the test property. Such an automaton represents a monitor of the satisfaction of the property during the execution of the system/model. It can be used to measure the coverage of the property by existing test suites, as explained in [7], but also to produce test cases that aim at exercising the property according to dedicated coverage criteria. We introduce, in a first contribution, dedicated property automata coverage criteria, inspired from classical automata coverage criteria, that aim at characterizing relevant tests highlighting the behaviours described in the property. In this context, the tests are directly extracted from the automaton as is (arrow 1), to cover the events of the property. As a second contribution, we propose to refine the property automaton so as to exhibit possible violations of the property, and derive test cases that aim at ensuring the robustness of the system w.r.t. the property. To this end, we apply dedicated mutation operators (arrow 2) on the transitions of the original automaton $A_{\varphi}$, producing a new automaton $A'_{\varphi}$ that is then exploited similarly.

This paper is organized as follows. Section 2 presents the considered UML subset that is used for our model-based testing approach and introduces a running example. Then, Section 3 presents the property formalization language and its associated semantics as automata. The first contribution, namely the property automata coverage criteria, is presented in Section 4. The second contribution, namely the mutation of automata transitions, is explained in Section 5. The experimental assessment of this approach is reported in Section 6. Then, Section 7 compares our approach with related works. Finally, Section 8 concludes and presents the future works.

2 UML/OCL Models and Running Example

This section presents the UML/OCL models that are considered in our approach, along with a running example that will be used to illustrate the contributions in the rest of the paper. As our approach is technically bound to the CertifyIt test generation engine, we focus on the subset of UML considered by this tool. However, the approach could be applicable to any other modelling language.

2.1 UML4ST – a subset of UML for Model-Based Testing

The UML models we consider are those supported by the CertifyIt test generator, commercialized by the Smartesting company. This tool automatically produces model-based tests from a UML model [4] with OCL code describing the behaviors of the operations. CertifyIt does not consider the whole UML notation as input, it relies on a subset named UML4ST (UML for Smartesting) which considers class
diagrams, to represent the data model, augmented with OCL constraints [23], to describe the dynamics of the system. It also requires the initial state of the system to be represented by an object diagram. Finally, a statechart diagram can be used to complete the description of the system dynamics.

OCL provides the ability to navigate the model, select collections of objects and manipulate them with universal/existential quantifiers to build first-order logic expressions. Regarding the OCL semantics, UML4ST does not consider the third logical value undefined that is part of the classical OCL semantics. All expressions have to be defined at run time in order to be evaluated. CertifyIt interprets OCL expressions with a strict semantics, and raises execution errors when encountering null pointers.

These restrictions w.r.t. the classical UML semantics originate from the fact that the UML/OCL model aims at being used for Model-Based Testing purposes. As such, it requires to use an executable UML/OCL model, since the abstract test cases are obtained by animating the model.

2.2 Running Example

We illustrate the UML/OCL models that are considered using a simple model of a web application named eCinema. This application provides a means for registered users to book tickets for movies that are screened in a cinema.

The UML class diagram, depicted in Fig. 2 contains the classes of the application: ECinema, Movie, Ticket and User. The ECinema class models the system under test (SUT) and contains the API operations offered by the application. This application proposes classical online booking features: a registered user may login to application, purchase tickets, view his basket, delete one or all tickets from his basket, and logout.

Figure 3 shows the OCL code of the buyTicket operation. Upon invocation, the caller provides the title of the movie for which (s)he wants to buy a ticket. The operation first checks that the user is registered on the application, and then checks if there exists an unallocated tickets for this movie. If all these verifications succeed, a ticket is associated to the user. This operation is specified in a defensive style, and as such, its precondition is always true, and the postcondition is in charge of distinguishing nominal cases from erroneous calls. We assume in the rest of the paper that all operations are specified this way, which is realistic since the model is used for testing purposes.

The OCL code of this operation contains non-OCL annotations, inserted as comments, such as ---@AIM: id and ---@REQ: id. The ---@AIM: id tags denote test targets while the ---@REQ: id tags mark requirements from the informal specifications. These tags can be used to reference a particu-
context ECinema::buyTicket(in_title : ECinema::TITLES): oclVoid

effect:
  ---@REQ: BASKET_MNGT/BUY_TICKETS
  if self.current_user.oclIsUndefined() then
    message = MSG::LOGIN_FIRST ---@AIM: BUY_Login_Mandatory
  else
    let tm: Movie = self.all_listed_movies->any(m: Movie | m.title = in_title) in
    if tm.available_tickets = 0 then
      message= MSG::NO_MORE_TICKET ---@AIM: BUY_Sold_Out
    else
      let t: Ticket = (Ticket.allInstances())->any(owner_ticket.oclIsUndefined()) in
      self.current_user.all_tickets_in_basket->includes(t) and
      tm.all_sold_tickets->includes(t) and
      tm.available_tickets = tm.available_tickets - 1 and
      message= MSG::NONE ---@AIM: BUY_Success
    endif
  endif

Figure 3: OCL code of the buyTicket operation

lar behaviour of the operation (e.g. @AIM:BUY_Login_Mandatory represents a failed invocation of this operation, due to the absence of a user logged on the system). Notice that it is possible to know which tags were covered during the execution of the model, inside the test cases, providing a feedback on the structural coverage of the OCL by the test cases.

3 Property Patterns and Automata

We now present the property language and their associated property automata.

3.1 Property Pattern Language

The property description language is a temporal extension of OCL. This language is based on patterns which avoids the use of complex temporal formalisms, such as LTL or CTL. We ground our work on the initial proposal of Dwyer et al. [12] in which a temporal property is a temporal pattern that holds within a scope. Thus, the user can define a temporal property choosing a pattern and a scope among a list of predefined schema. The scopes are defined from events and delimit the impact of the pattern. The patterns are defined from events and state properties to define the execution sequences that are correct. The state properties and the event are described based on OCL expressions.

Patterns. There are five main temporal patterns: (i) always $P$ means that state property $P$ is satisfied by any state, (ii) never $E$ means that event $E$ never occurs, (iii) eventually $E$ means that event $E$ eventually occurs in a subsequent state, this pattern can be suffixed by a bound which specifies how many occurrences are expected (at least $k$, at most $k$, exactly $k$ times), (iv) $E_1$ (directly) precedes $E_2$ means that event $E_1$ (directly) precedes event $E_2$, (v) $E_1$ (directly) follows $E_2$ means that event $E_2$ is (directly) followed by event $E_1$.

Scopes. Five scopes can apply to a temporal pattern $TP$: (i) $TP$ globally means that $TP$ must be satisfied on any state of the whole execution, (ii) $TP$ before $E$ means that $TP$ must be satisfied before the first occurrence of $E$, (iii) $TP$ after $E$ means that $TP$ must be satisfied after the first occurrence of $E$, (iv) $TP$ between $E_1$ and $E_2$ means that $TP$ must be satisfied between any occurrence of $E_1$ followed by an occurrence of $E_2$, (v) $TP$ after $E_1$ until $E_2$ means that $TP$ must be satisfied between any occurrence of $E_1$ followed by an occurrence of $E_2$ and after the last occurrence of $E_1$ that is not followed by an occurrence of $E_2$. 
**Events.** Scopes and patterns refer to events that can be of two kinds. On the one hand, events denoted by `iscalled(op, pre, post, {tags})` represent operation calls. In this expression, `op` designates the operation name, `pre` and `post` are OCL expressions respectively representing a precondition and a postcondition. Finally, `{tags}` represents a set of tags that can be activated by the operation call. Such an event is satisfied on a transition when the operation `op` is called from a source state satisfying the precondition `pre` and leading to a target state satisfying the postcondition `post` and executing a path of the control flow graph of the operation `op` which is marked by at least one tag of the set of tags denoted `{tags}`. Notice that the three components `pre`, `post` and `{tags}` are optional. On the other hand, events denoted by `becomesTrue(P)` where `P` is an OCL predicate, are satisfied by any operation call from a state in which `P` evaluated to false, reaching a state in which `P` evaluates to true.

**Example 1 (Property Example)** Consider the eCinema example given in Sect. 2.2. An informal access control requirement expresses that: "the user must be logged on the system to buy tickets". This can be expressed by the following three properties that put the focus on various parts of the execution of the system.

\[
\begin{align*}
\text{never} & \quad \text{iscalled(buyTicket, \{@AIM:BUY_Success\})} \\
\text{before} & \quad \text{iscalled(login, \{@AIM:LOG_Success\})} \quad (\text{Property 1}) \\
\text{eventually} & \quad \text{iscalled(buyTicket, \{@AIM:BUY_Success\}) \text{ at least 0 times}} \\
\text{between} & \quad \text{iscalled(login, \{@AIM:LOG_Success\})} \\
\text{and} & \quad \text{iscalled(logout, \{@AIM:LOG_Logout\})} \quad (\text{Property 2}) \\
\text{never} & \quad \text{iscalled(buyTicket, \{@AIM:BUY_Success\})} \\
\text{after} & \quad \text{iscalled(logout, \{@AIM:LOG_Logout\})} \quad (\text{Property 3}) \\
\text{until} & \quad \text{iscalled(login, \{@AIM:LOG_Success\})}
\end{align*}
\]

First, with Property (1), we specify that before a first successful login, it is not possible to succeed in buying a ticket. Second, we specify that when the user is logged in, he may buy a ticket (Property (2)). Notice that this property uses a workaround of the eventually pattern to express an optional action. Finally, Property (3) specifies that it is also impossible to buy a ticket after logging out, unless logging in again.

### 3.2 Property Semantics using Automata

The properties are interpreted on executions that are viewed as sequences of pairs of a state and an event that represent a sequence of transitions. The semantics of the test properties are expressed by means of automata. Indeed, the temporal language is a linear temporal logic whose expression power is included in the \(\omega\)-regular languages.

The semantics of a temporal property is a labelled automaton which is defined by Def. 1. The method that associates an automaton to a temporal property is completely defined in [6]. This automaton describes the set of accepted executions of the property and highlights specific transitions representing the events used in the property description. In addition, the automaton may contain at most one rejection state that indicates the violation of the property when reached.

**Definition 1 (Property Automaton)** Let \(\Sigma\) be a set of events. An automaton is a quintuplet \(\mathcal{A} = (Q, q_0, F, R, T)\) in which: \(Q\) is a finite set of states, \(q_0\) is an initial state \((q_0 \in Q)\), \(F\) is a set of final states \((F \subseteq Q)\), \(R\) is a set of rejection states \((R \subseteq Q)\), \(T\) is a set of transitions \((T \subseteq Q \times \mathcal{P}(\Sigma) \times Q)\) labelled by a set of events.
We call $\alpha$–transitions the transitions of $T$ that are labelled by the events expressed in the original property, and we call $\Sigma$–transitions the other transitions. $\Sigma$–transitions are named after their expression as they are labelled by a restriction on $\Sigma$ (the set of all possible events).

Notice that, when considering safety properties (something bad never happens), the set $R$ of rejection state is necessarily not empty. Notice also that, for a given state (resp. transition), it is possible to know if the state (resp. transition) originates from the scope or the pattern of the property. Notice also that the final states catch that the scope has been executed at least once. Thus final states are not accepting states as in traditional Büchi automata; they represent the test goals in the sense that we expect test cases to reach such states at some point.

Events in the automaton are quadruplets $[op, pre, post, \{tags\}]$ in which $op$ designates an operation, $pre$ and $post$ respectively denote pre- and postconditions, and $tags$ specifies a set of tags. The events used in the test properties are thus rewritten to match this formalism: $\text{isCalled}(op, pre, post, \{tags\})$ rewrites to $[op, pre, post, \{tags\}]$ and $\text{becomesTrue}(P)$ rewrites to $[\_, not(P), P, \_]$, in which $\_$ replaces any acceptable value of the corresponding component.

**Example 2 (Automaton of a Temporal Property)** Consider the property given in Example 1. Figure 4 shows the automata representation associated to Property 1 (left), Property 2 (middle) and Property 3 (right). Notice that the left-hand side (resp. right-hand side) automaton displays an error state, identified by “X”, that can be reached if the system authorizes to perform a ticket purchase before logging in (resp. after logging out and before logging in again). The automaton of Property 2, in the middle, does not display an error state meaning that the property can never be falsified. In addition, it exhibits a reflexive transition that represents the optional event, that may or may not take place when the user is logged.

On these automata, the $\alpha$--transitions are the transitions labelled by events $E0$, $E1$ and $E2$. The other transitions are $\Sigma$--transitions.

These automata provide a means for monitoring the satisfaction of the property by the execution of the model. We assume that the model is correct w.r.t. the property. In that sense, only valid traces can be extracted from the model, and no transition leading to an error state can possibly be activated.

The next two sections show how to exploit these automata by defining dedicated test coverage criteria, that can be used either for evaluating an existing test suite, or for generating new tests supposed to

![Figure 4: Automata representation for the properties given in Example 1](image-url)
illustrate or exercise the property.

4 Property Automata Coverage Criteria

We present in this section the automata coverage criteria that we propose. These dedicated coverage criteria originate from the observation that classical coverage criteria on automata are not relevant for our property automata. Indeed, criteria such as transition coverage, or transition-pair coverage, make no distinction between the transitions of the automata. However, in our case, all the transitions of the property automata are not of equal importance. For example, consider the automata provided in Fig. 4. Reflexive $\Sigma -$transitions only exist to capture all possible executions of the model but their sole purpose is to make it possible to put the model in a state from which the $\alpha -$transitions can be activated. While classical coverage criteria would target these $\Sigma -$transitions, we propose new coverage criteria focused on $\alpha -$transitions, aiming at activating them, but also focusing on different paths which iterate over specific parts of the automaton. We present these criteria and illustrate, for each of them, their relevance in terms of property coverage. Before that, we start by introducing some preliminary definitions.

4.1 Preliminary Definitions

We consider that an abstract test case is defined on the model as a finite sequence of steps, each of them formalized by \( s_{i+1}, \bar{o}_i, tags_i \leftarrow op_i(\bar{in}_i, s_i) \) (for \( i \geq 0 \) and \( i < \) the length of the test case) in which \( s_i \) (resp. \( s_{i+1} \)) is the model state before (resp. after) the step, \( op_i \) is a model operation called with inputs \( \bar{in}_i \) returning outputs \( \bar{o}_i \) and activating the behaviours identified by the \( tags_i \) set. We denote by \( s_0 \) the initial state of the model.

The conversion of a test case (computed from the model) into a path of the automaton is made by matching the steps of the test case with the events of the automaton, accordingly to the following definition.

Definition 2 (Step/Event matching) A step formalized by \( s_{i+1}, \bar{o}_i, tags_i \leftarrow op_i(\bar{in}_i, s_i) \) is said to match an event \([op, pre, post, tags]\) if and only if the four conditions hold: (i) \( op = op_i \) or \( op \) is undefined (symbol $\bot$), (ii) \( pre \) is satisfied in \( s_i \) (modulo substitution of \( \bar{in}_i \) in \( pre \)), (iii) \( post \) is satisfied in \( s_{i+1} \), and (iv) \( tags \cap tags_i \neq \emptyset \)

Given a test case, each step \( s_{i+1}, \bar{o}_i, tags_i \leftarrow op_i(\bar{in}_i, s_i) \) is matched against the possible transitions \( q \rightarrow q' \) that can be activated from the current automaton state \( q \) (initially, \( q_0 \) when the first step is considered). When a given step/event is matched, the exploration of the automaton restarts from \( q' \) the state targeted by the transition. As the property automata are deterministic and complete, there is exactly one transition that can be matched by each step of the test case.

4.2 Coverage Criteria for the Property Automata

We present in this section the four coverage criteria that we propose. Since the model is expected to satisfy the property, the paths inescapably leading to the error state are not supposed to be coverable, and thus, their transitions are not considered in the coverage criteria that we now present.

The first two coverage criteria that we propose consider the $\alpha -$transitions.

Definition 3 ($\alpha -$transition coverage) A test suite is said to satisfy the $\alpha -$transition coverage criterion if and only if each $\alpha -$transition of the automaton is covered by at least one test case of the test suite.
This first coverage criterion is an adaptation of the classical transitions-coverage criteria from the literature [16]. It aims at covering the transitions that are labelled by events initially written in the associated temporal property. A test suite satisfying this criterion ensures that all the events expressed at the property level are highlighted by the test suite.

**Example 3 (α-transition coverage)** On the example shown in Fig. 4, for Property 2, a test suite satisfying the α-transition coverage criterion ensures that at least one test case illustrates the optional ticket purchase by covering transition $E_1 0 \rightarrow 1$. Also, another test case should illustrate the fact that two iterations of the scope are possible, by covering transition $E_0 1 \rightarrow 2$.

**Definition 4 (α-transition-pair coverage)** A test suite is said to satisfy the α-transition-pair coverage criterion if and only if each successive pair of α-transitions is covered by at least one test case of the test suite.

Notice that this criterion considers the coverage of pairs of α-transitions reaching a particular state, and originating from the same state. However, it is possible to display intermediate Σ-transitions between a pair of α-transitions.

**Example 4 (α-transition-pair coverage)** On the example shown in Fig. 4, for Property 2, a test suite satisfying the α-transition coverage criterion ensures the coverage of the following pairs:

- $(0 0 \rightarrow 1, 1 0 \rightarrow 1)$
- $(1 0 \rightarrow 1, 1 E_1 0 \rightarrow 2)$
- $(1 E_1 0 \rightarrow 2, 2 0 \rightarrow 1)$
- $(2 E_0 1 \rightarrow 1, 1 E_1 0 \rightarrow 1)$
- $(2 E_0 1 \rightarrow 1, 1 E_2 0 \rightarrow 2)$

A test suite satisfying this coverage criterion thus ensure the existence of tests illustrating the buying of a ticket, but also tests performing a login followed by a logout without intermediate ticket purchase, and also tests illustrating the optional ticket purchase in a second iteration over the scope.

The last two coverage criteria that we propose consider the structure of the property and aim at covering internal or external loops inside the property automaton, in order to iterate over the pattern or the scope of the property.

**Definition 5 (k-pattern coverage)** A test suite is said to satisfy the k-pattern-activation coverage criterion if and only if the α-transitions of the pattern of the automaton are iterated between 0 and k times, each loop in the pattern being performed without exiting the pattern-part of the automaton.

This coverage criterion aims at activating the internal loops inside the pattern-part of the automaton, without covering any transition of scope during these iterations. This coverage criterion is not applicable to any pattern; it only applies to precedes, follows and some forms of the eventually pattern.

**Example 5 (k-pattern-coverage)** On the example shown in Fig. 4, for Property 2, a test suite satisfying the 2-pattern coverage criterion ensures the coverage of 0, 1, and 2 iterations of the reflexive α-transition $E_1 0 \rightarrow 1$.

**Definition 6 (k-scope coverage)** A test suite is said to satisfy the k-scope-activation coverage criterion if and only if the α-transitions of the scope of the automaton are iterated between 1 and k times, and covering each time at least one α-transition of the pattern.

This coverage criterion aims at activating the external loops outside the pattern-part of the automaton. Similarly to the k-pattern criterion, the k-scope criterion is not applicable to all scopes, but restricts its usage to only repeatable ones, namely between and after.
Example 6 (k-scope coverage) On the example shown in Fig. 4, for Property 2, a test suite satisfying the 2-scope coverage criterion ensures the coverage of 1 and 2 logout-login sequences, by covering cycle $E_1 \rightarrow 2 \rightarrow E_0 \rightarrow 1$.

These four coverage criteria are based on the property automata as is, and thus, will only illustrate the property and show that they are correctly implemented (e.g. the occurrences of events are authorized by the implementation, along with the repetition of scopes, etc.) However, showing that unexpected events do not appear requires an additional and dedicated strategy for robustness testing, that we now present.

5 Property Automata Mutation for Testing Robustness

The automata coverage criteria described in the previous section focus on activating events expressed within the test properties. Thus, these coverage criteria aim at illustrating that the properties are correctly implemented. However, in the cases of safety properties (something bad should never happen), it might be interesting to produce test cases that aim at an attempt to violate the property. The cases in which the property is violated are clearly identified in the automaton, being displayed through error states. Unfortunately, targeting the activation of transitions leading to these error states is irrelevant: since the model (used to compute the tests) is supposed to satisfy the property, these transitions can not be activated as is.

In this section, we propose mutation operators that apply to events labelling the transition leading to error states, so as to make them activable, thus providing an interesting test target that aims at leading to a possible violation of the property. These mutations and the robustness coverage criterion aims to simulate an erroneous implementation of the property that would mistakenly allow the activation of an unexpected event. As we are performing a Model-Based Testing approach, it is thus mandatory to be able to compute a sequence performing forbidden events, at the model level.

5.1 Event Mutation Operators

Our goal is to provoke unexpected events. As these latter can not be activated on the model, the idea is to get closer to the inactivable event. To achieve that, we apply mutations on these events. These mutations apply mainly to the uncontrollable part of the events (postconditions and tags), and keep the controllable part the lesser modified.

The mutations we propose modify the transitions of the automata. They target the events labelling the transitions, and can be of two kinds: (i) predicate mutation rules, inspired from classical mutation operators over predicates [11], applied to pre- and postconditions, and (ii) tag mutation rules applied to the tag list of the events.

Postcondition/Tag Removal. This rule consists in removing the postcondition and the tag list from the event.

$$[op, pre, post, T] \sim [op, pre, \_\_]$$

Both tags and postconditions are systematically removed, as these two elements are frequently related. Their combined removal thus avoids creating inactivable events.
Precondition Removal. This rule consists in removing the precondition of the event.

\[ [op, pre, post, T] \leadsto [op, \ldots, \ldots] \]

When applied, this mutation also removes the postcondition and tags, in order to weaken the event, and increase the chances that the mutation will produce an activable event.

Predicate Weakening. The predicate removal mutation replaces each literal in a conjunction by true. This removal applies to both pre- and postconditions.

\[ [op, A \land B, C \land D, T] \leadsto [op, A, \ldots, \ldots], [op, B, \ldots, \ldots], [op, A \land B, C, \ldots], [op, A \land B, D, \ldots] \]

When applied to the postcondition, this rule removes the tags from the event. If it is applied to the precondition, this rule also removes the postcondition from the event.

Example 7 (Event mutation) Consider the examples provided on Fig. 4, left-hand side or right-hand side. In both cases, event \( E_1 = [\text{buyticket, \ldots, \ldots}, \{\text{@AIM:BUY_Success}\}] \) can be rewritten to \( E'_1 = [\text{buyticket, \ldots, \ldots}] \). This event represents the attempt to perform a ticket purchase but without any expectation regarding the success or the failure of this operation.

5.2 Automata Mutation and Robustness Coverage Criteria

The mutation operators that we propose can be applied on a given property automaton \( \mathcal{A} \). The automaton is modified as follows: (i) each transition leading to the error state is mutated, and (ii) the targeted error state becomes the only final state of the new automaton. We denote \( \mathcal{A}' \) the new automaton obtained after mutation.

Example 8 (Automaton mutation) Figure 5 displays the application of a mutation on the automaton associated to Properties (1) and (3). We see that the mutated automaton makes it possible to match test cases that would perform an attempt to purchase a ticket, before successfully logging in.

Definition 7 (Robustness coverage) A test suite is said to satisfy the robustness coverage criterion for a property \( P \) if and only if the mutated transition of each mutated automaton of property \( P \) is covered by at least one test case of the test suite.

Example 9 (Robustness coverage) In order to activate the mutated event of Property 1, and thus, check the robustness of the system w.r.t. it, the validation engineer can design the following test case:

\[ E_1: [\text{buyticket, \ldots, \ldots}, \{\text{@AIM:BUY_Success}\}] \leadsto E'_1: [\text{buyticket, \ldots, \ldots}] \]

Figure 5: Mutation of the automaton for Property 1 (left) and Property 3 (right)
<table>
<thead>
<tr>
<th>Step</th>
<th>Operation</th>
<th>Expected behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>sut.buyTicket(TITLE1)</code></td>
<td>@AIM:BUY_Error_Login_First</td>
</tr>
<tr>
<td>2</td>
<td><code>sut.login(REGISTERED_USER,REGISTERED_PWD)</code></td>
<td>@AIM:LOG_Success</td>
</tr>
</tbody>
</table>

On a correct implementation, the system should not allow the first operation (`buyTicket`) to be performed successfully. If the implementation conforms to the model, then it is expected to activate an erroneous behavior of this operation (as predicted by the model). If the implementation is incorrect, the `buyTicket` operation will return successfully and thus display a behavior that differs from the model.

In order to activate the mutated event of Property 3, the validation engineer can design the following test case:

<table>
<thead>
<tr>
<th>Step</th>
<th>Operation</th>
<th>Expected behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>sut.login(REGISTERED_USER,REGISTERED_PWD)</code></td>
<td>@AIM:LOG_Success</td>
</tr>
<tr>
<td>2</td>
<td><code>sut.logout()</code></td>
<td>@AIM:LOGLogout</td>
</tr>
<tr>
<td>3</td>
<td><code>sut.buyTicket(TITLE1)</code></td>
<td>@AIM:BUY_Error_Login_First</td>
</tr>
</tbody>
</table>

Similarly, on a correct implementation, the last operation (`buyTicket`) should not succeed (as on the model). An incorrect implementation would allow this operation to be performed successfully.

We now present the experimental assessment of the approach proposed in this paper.

## 6 Experimentation

Our approach has been implemented into a specific framework [10], and applied in an industrial context during two national projects. This framework allows the user to write properties, measure the coverage of a property with an existing test suite, and generate test scenarios to satisfy a given (selected) coverage criterion\(^1\). We report here the usage of the tool and its evaluation by test engineers. In addition, we present an experimental evaluation of the capabilities of our tool in terms of fault detection, especially focusing on the robustness testing criteria.

**Experimentations during industrial projects.** Our approach has been initially designed during the ANR TASCCC project\(^2\), and implemented into a prototype of the same name. This project was done in collaboration with (among others) the Smartesting company (technology provider with the CertifyIt test generator), and Gemalto (case study provider) and focused on the validation of smart card products for Common Criteria evaluations. In this context, the properties we designed aimed at expressing functional security properties on a smart card case study. It has also been exploited during the ANR OSeP project\(^3\), also in partnership with the Smartesting company, and funded by the Armaments Procurements Agency. This project focused on online and offline techniques for testing security (cryptographic) components. In both projects, we had the opportunity to address the following three questions: *How easy it is to learn and use the test property language?* (Q. 1) *How does this approach compare to a functional approach such as Smartesting CertifyIt?* (Q. 2) *What is the interest of our industrial partners in our property-based testing approach?* (Q. 3)

For our experiments, we considered the case studies developed by our partners. We started by designing test properties for the three considered case studies. The language turned out to be easy to learn, as the number of combinations is relatively small. With a little help from us, the validation engineers were

---

\(^1\)see http://vimeo.com/53210102 for a video demo of the tool  
\(^2\)http://lifc.univ-fcomte.fr/TASCCC  
\(^3\)http://osep.univ-fcomte.fr
able to express some security properties in natural language, rephrase them in order to fit the allowed combinations of scopes/patterns, and choose the appropriate entities to express these properties [Q. 1]. However, we noticed some pitfalls in its use. Indeed, the validation engineers tend to express test cases with the properties, e.g. by specifying sequences of events without taking care of the applicability of the sequence in general.

For each case study, our industrial partners generated functional test cases using the Smartesting CertifyIt test generator. Thus, we measured the coverage of the test properties that were designed. For each of the three considered case studies, we noticed that some properties where not covered at all (no \( \alpha \)-transition of their associated automaton were activated), meaning that the test cases do not activate the scope of the property [Q. 2]. Such a process provides an interesting feedback on the coherence of the model w.r.t. the properties and the completeness of the test suite. When a given coverage criterion is not satisfied, the validation engineer has the possibility to see which parts of the automaton were not covered, and focus his effort on them [Q. 3]. In some cases, we detected some violations of the property during the test cases executions. In this case, the coverage report shows that transitions leading to the error state of the property automaton is reached. This indicates that the model does not satisfy the property. There are two reasons for that, either the property is too restrictive (e.g. in the cases described above), or the model is incorrectly designed (this was never the case). It is then mandatory to correct one or the other of these artefacts. In this end, this process can be (indirectly) used to validate the model, by checking that it behaves as described in the properties.

In addition, we asked our industrial partners to evaluate the relevance of the produced tests (and, by extension, the relevance of the proposed coverage criteria). They pointed out the interest of the graphical visualization of the property automaton which shows, and addresses, various configurations that one might forget when designing tests manually.

**Evaluation of fault detection capabilities.** For our experiments, we have considered the eCinema model, along with additional properties that aim at illustrating various combinations of scopes and patterns. In addition to properties (1)-(3) presented before, we considered three more specifying that: “before a successful ticket deletion, there eventually exists at least one successful ticket buying” (4), “globally, a successful login precedes a successful logout” (5), and “after having deleted all the tickets, it is not possible to delete one ticket unless buying one in the meantime” (6).

We started by generating a test suite with the Smartesting CertifyIt test generator, and completed this test suite with 11 tests in order to satisfy the different coverage criteria for each properties. We do not detail this process, as it is not the purpose of this paper; however, the interested reader may refer to [10]. We then mutated the original eCinema model, by blindly using the following mutation operators, dedicated to OCL code: Simple Set Operator Replacement (SSOR) replacing a set operator by another, Simple expression Negation Operator (SNO) negating an atomic condition inside a predicate, Stuck-At-False (SAF) replacing predicates by false, and Action Deletion (AD) deleting a predicate of the postcondition (e.g. to remove a variable assignment).

We then run our test suite on these mutant models. We compared, at each step of each test, the expected result of each operation (given by the test case) with the actual returned values (this is how conformance is established on a concrete implementation), giving us a conformance verdict. In addition, we monitored the execution of each test on the automata associated to the considered test properties, to check if the observation was too weak to detect an actual property violation. Finally, we compared our test suite with the original test suite computed by CertifyIt. Figure 6 show the results we obtained. For each considered mutation operator we provide the overall number of mutants that were detected as: conform and not reaching an error state on the automaton (C-NE – it is either an equivalent mutant or
a mutant that violates the property but it was not possible to observe it), non-conform but not reaching an error state (NC-NA – it is the cases of mutants that are not related to the property or, at least, not killed for that reason), non-conform and reaching an error state (NC-E – actual violations of the property detected using basic observations), conform but reaching an error state (C-A – actual violations of the property that have not been detected using basic observations, but could be detected by more intrusive means such as monitoring).

Experimental results show that our technique is able to: (1) build test cases that consist in operations leading to a violation of the property (see lines SSOR and AD in which respectively 2 and 4 tests detect the mutant using the property automaton), (2) build test cases that make property violations observable (see line SNO), and (3) build new test cases that are likely to reveal other non-conformances (even if they are not related to the property), improving the overall efficiency of the test suite (see line AD).

7 Related Work

The notion of property-based testing is often employed in the test generation context. Several approaches [15, 21, 1] deal with LTL formulae, that are negated and then given to a model-checker that produces traces leading to a counter-example of this property, and thus defining the test sequences. Our work improves these approaches by defining both nominal and robustness test cases, aiming either at illustrating the property or checking the system’s robustness w.r.t. it. A recent work [14] defines the notion of property relevant test cases, introducing new coverage criteria that can be used to determine positive and negative test cases. Nevertheless, our approach proposes several differences. First, we do not rely on LTL, but on a dedicated language easier to manipulate than LTL by non-specialists. Second, the notion of property-relevance is defined at the LTL level, whereas we rely on the underlying automata. Finally, the relevance notion acts as an overlay to classical coverage criteria, while we propose new ones.

Based on Dwyer’s work, jPost [13] uses a property expressed in a trace logic for monitoring an implementation. Similarly, in [19] the authors introduce the notion of observers, as ioSTS, that decide the satisfaction of the property and guide the test generation within the STG tool. Our work differs in the sense that the coverage criteria are not only used as monitors for passive testing, but they can also be employed for active testing.

A lot of scenario-based testing works focus on extracting scenarios from UML diagrams, such as the SCENTOR approach [24] or SCENT [20] using statecharts. The SOOFT approach [22] proposes an object oriented framework for performing scenario-based testing. In [3], Binder proposes the notion of round-trip scenario test that cover all event-response path of an UML sequence diagram. Nevertheless, the scenarios have to be completely described. Our approach proposes to automatically generate the test scenarios from higher level descriptions of the properties the validation engineer wants to test. In [2], the authors propose an approach for the automated scenario generation from environment models for testing of real-time reactive systems. The behavior of the system is defined as a set of events. The process relies on an attributed event grammar (AEG) that specifies possible event traces. Even if the targeted

<table>
<thead>
<tr>
<th>Test suites</th>
<th>Property-Based Testing</th>
<th>Smartesting CertifyIt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mutations / Verdicts</td>
<td>C-NE</td>
<td>NC-NA</td>
</tr>
<tr>
<td>SSOR</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>SNO</td>
<td>28</td>
<td>2</td>
</tr>
<tr>
<td>SAF</td>
<td>31</td>
<td>1</td>
</tr>
<tr>
<td>AD</td>
<td>6</td>
<td>12</td>
</tr>
</tbody>
</table>

Figure 6: Results of the mutant detection for the two considered test suites
applications are different, the AEG can be seen as a generalization of regular expressions. Our approach goes further as it uses a property description language that is close to a natural language.

8 Conclusion and Future Works

In this paper, we have presented a model-based testing process based on test properties. These latter are expressed in a dedicated formalism that captures the dynamics of the system. Each property is translated into an automaton, for which new coverage criteria have been introduced, in order to illustrate the property. In addition, we propose to refine the automaton so as to exhibit specific transitions that are closely related to error traces that are not accepted by the property. This technique makes it possible to introduce a notion of robustness testing to ensure that the property is correctly implemented. We have tool-supported this approach to apply it to UML/OCL models [10]. The advantages of this approach are twofold. Mainly, it provides a means to produce test cases that can be directly related to the property. Such a traceability makes it a suitable approach for industrial purposes. In addition, the automata and their refinements can be used to measure the coverage of corner cases of a property for an existing test suite. This approach has been evaluated in the context of industrial projects, which gave us a very positive feedback on the usefulness of the coverage criteria, exhibiting specific sequences of operations one may want to consider when testing. Finally, notice that the proposed coverage criteria are not specific to UML/OCL and could be adapted to any other notation that would use the same notions of scope and patterns with a different representation of events.

For the future, we first plan to extend our property language to introduce local variables. Such an extension would greatly improve the expressiveness of our property language. However, this extension would imply the definition of data coverage criteria dedicated to the coverage of the properties of these values. Second, we are also investigating a way to efficiently generate the tests satisfying the coverage criteria we proposed. For now, a solution using test scenarios has been implemented. However, the combinatorial unfolding of the scenarios compromises the full automation of the test generation approach.

References


Spinal Test Suites for Software Product Lines

Harsh Beohar  
Center for Research on Embedded Systems  
Halmstad University, Sweden  
harsh.beohar@hh.se

Mohammad Reza Mousavi  
Center for Research on Embedded Systems  
Halmstad University, Sweden  
m.r.mousavi@hh.se

A major challenge in testing software product lines is efficiency. In particular, testing a product line should take less effort than testing each and every product individually. We address this issue in the context of input-output conformance testing, which is a formal theory of model-based testing. We extend the notion of conformance testing on input-output featured transition systems with the novel concept of spinal test suites. We show how this concept dispenses with retesting the common behavior among different, but similar, products of a software product line.

1 Introduction

1.1 Motivation

Testing and debugging are labor-intensive parts of software development. In particular, testing a software product line is extremely time- and resource-consuming due to the various configurations of products that are derivable from the product line. In order to manage the complexity, the test process of a software product line must be efficiently coordinated: common features ought to be tested once and for all and only specific variation points of various configurations should be tested separately.

Model-based testing is an approach to structure the test process by exploiting test models. Input-output conformance testing (ioco) [24] is a formalization of model-based testing employing input-output labeled transition systems as models. In the past, we extended the formal definition ioco to the setting of software product lines [3], by exploiting input-output featured transition systems. In this paper, we define a theoretical framework, which serves as the first step towards an efficient discipline of conformance model-based testing for software product lines.

To this end, we define the notion of spinal test suite, which allows one to test the common features once and for all, and subsequently, only focus on the specific features when moving from one product configuration to another. We show that spinal test suites are exhaustive, i.e., reject each and every non-conforming implementation under test, when the implementation satisfies the orthogonality criterion. This is a rather mild criterion, which implies that old features are not capable of disabling any enabled behavior from the new features on their own and without involving any interaction with the new feature’s components.

1.2 Running example

To motivate various concepts throughout the paper, we use the following running example. Consider an informal description of a cruise controller, present in contemporary cars. The purpose of a cruise controller is to automatically maintain the speed of the car as specified by the driver. We denote the basic feature of a cruise controller by cc. Cruise controllers also have an optional feature, called collision avoidance controller (cac), whose task is to react to any obstacle detected ahead of the car within a danger...
zone. In case the collision avoidance feature is included in a cruise controller and an obstacle is detected, the engine power is regulated using an emergency control algorithm.

1.3 Organization

The rest of this paper is structured as follows. In Section 2, we recall the formal definitions regarding models, product derivation and conformance testing. In Section 3, we define the notion of spinal test suite, which is a compact test suite for the “new” features with respect to an already tested product (or a set of features). In Section 4, we study the exhaustiveness of the spinal test suites: we show that spinal test suites are in general non-exhaustive, but this can be remedied by requiring mild conditions on the implementation under test. In Section 5, we sketch the context of this research. In Section 6, we conclude the paper and outline the direction of our ongoing research.

2 Background

2.1 Input-output featured transition systems

Feature diagrams [13, 22] have been used to model variability constraints in SPLs using a graphical notation. However, it is well known that feature diagrams only specify the structural aspects of variability and they should be complemented with other models in order to specify the behavioral aspects [7]. To this end, we describe the behavior of a software product line using an input-output featured transition system (IOFTS) [3], defined and explained below.

Let $F$ be the set of features (extracted from a feature diagram) and $B = \{\top, \bot\}$ be the set of Boolean constants; we denote by $B(F)$ the set of all propositional formulae generated by interpreting the elements of the set $F$ as propositional variables. For instance, in our running example, formula $cc \wedge \neg cac$ asserts the presence of cruise controller and the absence of collision avoidance controller. We let $\phi, \phi'$ range over the set $B(F)$.

**Definition 1.** A input-output featured transition system (IOFTS) is a 6-tuple $(S, s, A_\tau, F, T, \Lambda)$, where

1. $S$ is the set of states,
2. $s \in S$ is the initial state,
3. $A_\tau = A_I \cup A_O \cup \{\tau\}$ is the set of actions, where $A_I$ and $A_O$ are disjoint sets of input and output actions, respectively, and $\tau$ is the silent (internal) action,
4. $F$ is a set of features,
5. $T \subseteq S \times A_\tau \times B(F) \times S$ is the transition relation satisfying the following condition (for every $s_1, s_2 \in S, a \in A_\tau, \phi, \phi' \in B(F)$):

$$\forall (s_1, a, \phi, s_2) \in T \land (s_1, a, \phi', s_2) \in T \Rightarrow \phi = \phi', \quad \text{\footnote{Here, by } \phi = \phi' \text{ \footnote{we assert that } \phi \text{ and } \phi' \text{ are syntactically equivalent.}}}$$

6. $\Lambda \subseteq \{\lambda : F \to B\}$ is a set of product configurations.

We write $s \xrightarrow{a}_{\phi} s'$ to denote an element $(s, a, \phi, s') \in T$ and drop the subscript $\phi$ whenever it is clear from the context. Graphically, we denote the initial state of an IOFTS by an incoming arrow with no
source state and we refer to an IOFTS by its initial state. Following the standard notation, we denote the reachability relation by \( \rightarrow \subseteq S \times A^* \times S \), which is inductively defined as follows:

\[
\begin{align*}
& s \xrightarrow{a} s' \xrightarrow{\tau} s'' \\
& s \xrightarrow{\alpha a} s'' \\
& s \xrightarrow{\sigma a} s''
\end{align*}
\]

Furthermore, the set of reachable states from a state \( s \) is denoted by \( \text{Reach}(s) = \{ s' | \exists \sigma s \xrightarrow{\sigma} s' \} \).

**Example 1.** Consider the IOFTS of a cruise controller, drawn in Figure 1, where inputs and outputs are prefixed with symbols \(?\) and \(!\), respectively. (Note that \(?\) and \(!\) are not part of the action names and are left out when the type of the action is irrelevant or clear from the context.) The regulate action, indicated by \( \text{rgl} \), regulates the engine power of the car when the cruise controller is activated. Furthermore, when \( \text{cac} \) is included in a product, some additional behavior may emerge. Namely, while the cruise controller is on, if an object is detected within a danger zone, then the cruise controller regulates the engine power in a safe manner denoted by \( \text{srgl} \). When the sensor signals a normal state, the cruise controller returns to the normal regulation regime. (For a realistic case study of a cruise controller and its formal model, we refer to [15].)

### 2.2 Product derivation operators

In [3], we introduced a family of product derivation operators (parameterized by feature constraints), which project the behavior of an IOFTS into another IOFTS representing a selection of products (a product sub-line).

**Definition 2.** Given a feature constraint \( \varphi \) and an IOFTS \( (S, s, A, F, T, \Lambda) \), the projection operator \( \Delta_\varphi \) induces an IOFTS \( (S', \Delta_\varphi(s), A_{\tau, \delta}, F, T', \Lambda') \), where

1. \( S' = \{ \Delta_\varphi(s') | s' \in S \} \) is the set of states,
2. \( \Delta_\varphi(s) \) is the initial state,
3. \( A_{\tau, \delta} = A_{\tau} \cup \{ \delta \} \) is the set of actions, where \( \delta \) is the special action label modeling quiescence [24],
4. \( T' \) is the smallest relation satisfying:

\[
\begin{align*}
& \exists \lambda \ ( \lambda \in \Lambda \land \lambda \models (\varphi \land \varphi')) \\
& \Delta_\varphi(s) \xrightarrow{a} \Delta_\varphi(s') \\
& \Delta_\varphi(s) \xrightarrow{\delta} \Delta_\varphi(s)
\end{align*}
\]

where the predicate \( Q(s, \lambda) \) is defined as

\[
\forall s', a, \varphi' \ (s \xrightarrow{a} s' \land a \in A_{O} \cup \{ \tau \}) \Rightarrow \lambda \models \varphi'.
\]
5. $\Lambda' = \{ \lambda \in \Lambda | \lambda \vdash \varphi \}$ is the set of product configurations.

In the above-given rules $\lambda \vdash \varphi$, denotes that valuation $\lambda$ of features satisfies feature constraint $\varphi$. Intuitively, rule (1) describes the behavior of those valid products that satisfy the feature constraint $\varphi$ in addition to the original annotation of the transition emanating from $s$. Rule (2) models quiescence (the absence of outputs and internal actions) from the state $\Delta_\varphi(s)$. Namely, it specifies that the projection with respect to $\varphi$ is quiescent, when there exists a valid product $\lambda$ that satisfies $\varphi$ and is quiescent, i.e., cannot perform any output or internal transition. Quiescence at state $s$ for a feature constraint $\lambda$ is formalized using the predicate $Q(s, \lambda)$, which states that from state $s$ there is no output or silent transition with a constraint satisfied by $\lambda$. In the conclusion of the rule, a $\delta$ self-loop is specified and its constraint holds when $\varphi$ holds and at least the feature constraint of one quiescent valid product holds. This ability to observe the absence of outputs (through a timeout mechanism) is crucial in defining the input-output conformance relation between a specification and an implementation [3].

Example 2. Consider the feature constraint $\varphi = cc \land \neg cac$. The IOFTS generated by projecting the IOFTS of cruise controller (in Figure 1) using feature constraint $\varphi$ is depicted in Figure 2. As mentioned before, this represents the product that has the basic cruise controller functionality but does not contain collision avoidance controller.

![Figure 2: Cruise controller after projecting with feature constraint cc \land \neg cac.](image)

In the sequel, we use the phrase “a feature specification $\Delta_\varphi(s)$” to refer to the following IOFTS:

$$(\text{Reach}(\Delta_\varphi(s)), \Delta_\varphi(s), A_\tau \delta, F, T, \Lambda).$$

We interpret the original IOFTS of Definition 1 as $\Delta_\top(s_0)$; this has the implicit advantage of always including quiescence in appropriate states.

2.3 Input-output conformance

The input-output conformance (ioco) testing theory [24] formalizes model-based testing in terms of a conformance relation between the states of a model (expressed as an input-output transition system) and an implementation under test (IUT). Note that the ioco theory is based on the testing assumption that the behavior of the IUT can be expressed by an input-output transition system, which is unknown to the tester.

The conformance relation can be checked by constantly providing the SUT with inputs that are deemed relevant by the model and observing outputs from the SUT and comparing them with the possible outputs prescribed by the model. In the following, we recall such an extensional definition of ioco, extended to software product lines in [3]. An equivalent intensional definition of ioco that relies on comparing the traces of the underlying IOFTS was also given in [3], but for the purpose of this paper we only work with the extensional definition. (After all, the extensional definition is the one that is supposed to be applied in practice.)
We begin with a notion of suspension traces generated by an IOFTS. Informally, a suspension trace is a trace that may contain the action $\delta$ denoting quiescence [24].

**Definition 3.** The set of suspension traces of a feature specification $\Delta_\varphi(s)$, denoted by $\text{Straces}(\Delta_\varphi(s))$, is defined as: \[ \{ \sigma \in A_\delta^* \mid \exists \psi \Delta_\varphi(s) \xrightarrow{\sigma} \Delta_\varphi(s') \} \].

For example, in the IOFTS of Example 2, $\delta$?on!rgl is a suspension trace emanating from the initial state $s_0$. Next, we define the notion of test suite, which summarizes all possible test cases that can be generated from a feature specification.

**Definition 4.** The test suite for an IOFTS $(\text{Reach}(\Delta_\varphi(s)), \Delta_\varphi(s), A \cup \delta, F, T, \Lambda)$, denoted by $\mathcal{T}(s, \varphi)$, is the IOFTS $(X \cup \{\text{pass}, \text{fail}\}, X_0, A_\delta, F, T', \Lambda)$, where

1. $X = \{ \{s' \mid \Delta_\varphi(s) \xrightarrow{\sigma} \Delta_\varphi(s')\}, \sigma \mid \sigma \in \text{Straces}(s)\}$ is the set of intermediate states and $\{\text{pass}, \text{fail}\}$ is the set of verdict states [24],
2. $X_0 = \{ \{s' \mid \Delta_\varphi(s) \xrightarrow{\sigma} \Delta_\varphi(s')\}, \varepsilon\}$ is the initial state of the test suite,
3. $A_\delta = A \cup \{\delta\}$ is the set of actions, and
4. the transition relation $T'$ is defined as the smallest relation satisfying the following rules.

\[
\begin{align*}
(X, \sigma), (Y, \sigma a) & \in X & (X, \sigma) \xrightarrow{a} \varphi (Y, \sigma a) \quad (3) \\
(X, \sigma) & \xrightarrow{a} \varphi \text{ pass} & (X, \sigma) \xrightarrow{a} \varphi \text{ pass} & (4) \\
\end{align*}
\]

\[
\begin{align*}
(a \in A_\delta \cup \{\delta\}) & \quad (X, \sigma) \xrightarrow{a} \varphi \text{ pass} & (X, \sigma) \xrightarrow{a} \varphi \text{ pass} & (5) \\
(X, \sigma) & \xrightarrow{a} \varphi \text{ fail} & (X, \sigma) \xrightarrow{a} \varphi \text{ fail} & (6) \\
\end{align*}
\]

Intuitively, the test suite for a feature specification is an IOFTS (possibly with an infinite number of states), which contains all the possible test cases that can be generated from the feature specification. Rule (3) states that if $X$ and $Y$ are nonempty sets of reachable states from $s$ (under feature restriction $\varphi$) with the suspension traces $\sigma$ and $\sigma a$, respectively, then there exists a transition of the form $(X, \sigma) \xrightarrow{a} \varphi (Y, \sigma a)$ in the test suite. Rules (4) and (5) model, respectively, the successful and the unsuccessful observation of outputs and quiescence. Note that input actions are not included in rules (4) and (5) because the implementation is assumed to be input-enabled [24]; hence, they are already covered by rule (3). Rule (6) states that the verdict states contain a self-loop for each and every output action, as well as for quiescence.

**Example 3.** The test suite for the IOFTS of Example 2 is (partially) depicted in Figure 3.

A reader familiar with the original IoCO theory [24] will immediately notice that our definition of a test suite (Definition 4) is nonstandard. In particular, a test suite is defined as a set of test cases (i.e., input-output transition systems with certain restrictions) with finite number of states in [24]; whereas we represent a test suite by an IOFTS, possibly with an infinite number of states. To this end, we define a test case to be a finite projection of a test-suite with the additional restriction that at each moment of time at most one input can be fed into the system under test (see [3] for a formal definition). As a result, our test cases are structurally similar to Tretmans’ formulation of the test cases, by which we mean that:

- a test case is always deterministic,
• a test case is always input enabled, and  
• a test case has no cycles except those in the verdict states pass and fail.

Another notable diﬀerence, that is key to deﬁne the concepts of Section 3, is that states of a test suite (or test case) carry some mathematical structure, whereas the states of a test case in [24] are abstract and carry no structure.

Next, we deﬁne a synchronous observation operator \( \models \) that allows us to model a test run on an implementation (cf. [24]). This is deﬁned over a test suite and an IOFTS (the intended implementation) as follows. (Note that the calligraphic letters \( X, Y \) in the following rules range over the states of a test suite.)

\[
\begin{align*}
X \overset{a}{\rightarrow} Y & \quad \Delta_{\varphi}(s) \overset{a}{\rightarrow} \Delta_{\varphi}(s') \quad a \in A_{\delta} \\
X \models \Delta_{\varphi}(s) & \quad Y \models \Delta_{\varphi}(s')
\end{align*}
\]

\[
\begin{align*}
\Delta_{\varphi}(s) \overset{\tau}{\rightarrow} \Delta_{\varphi}(s') \\
X \models \Delta_{\varphi}(s) & \quad Y \models \Delta_{\varphi}(s')
\end{align*}
\]

Having deﬁned the notion of synchronous observation, we can now deﬁne what it means for a feature speciﬁcation to pass a test suite. Informally, a feature speciﬁcation passes a test suite if and only if no trace of the synchronous observation of the test suite and the feature speciﬁcation leads to the fail verdict state.

**Deﬁnition 5.** Let \( X_0 \) be the initial state of a test suite \( T(s, \varphi) \). A feature speciﬁcation \( \Delta_{\varphi'}(s') \) passes the test suite \( T(s, \varphi) \) if

\[
\forall a \in A_{\varphi'} \quad X_0 \models \Delta_{\varphi'}(s') \quad X \models \Delta_{\varphi'}(s') \Rightarrow X \neq \text{fail}.
\]

The implementation \( \Delta_{\varphi'}(s') \) conforms to the speciﬁcation \( \Delta_{\varphi}(s) \) if \( \Delta_{\varphi'}(s') \) passes the test suite \( T(s, \varphi) \).

## 3 Spinal test suite

As mentioned in the introduction, one of the challenges in testing a software product line is to minimize the test eﬀort. The idea pursued in this section is to organize the test process of a product line incrementally. This is achieved by reusing the test results of an already tested product to test a product with
similar features, thereby dispensing with the test cases targeted at the common features. To this end, we introduce the notion of spinal test suite, which prunes away the behavior of a specified set of features from an abstract test suite $\mathcal{T}(s, \varphi)$ with respect to a concrete test suite $\mathcal{T}(s, \lambda)$ of the already tested product $\lambda$; the spinal test suite is only defined when $\lambda$ is valid w.r.t. $\varphi$, i.e., $\lambda \models \varphi$. The latter constraint means that the concrete product builds upon the already-tested features in the abstract test suite.

Notably, which behavior has to be pruned from an abstract test suite is crucial in defining a spinal test suite. One way to address this situation is by allowing only those reachable states in the abstract test suite from which a new behavior relative to the already tested product emanates. However, without any formal justification, we claim that such a strategy will not reduce the effort to test new behavior with respect to the already tested product.

For example, consider the test suite depicted in Figure 3 and suppose we have already tested the cruise controller without collision avoidance feature and now are interested in the correct implementation of the collision avoidance controller. Without collision avoidance feature, we know that there exists a product configuration $\lambda$ that is “new” with respect to the tested product $\lambda$. In-
3. The set of product configurations \( \Lambda' = \Lambda \setminus \{ \lambda \} \).

Intuitively, Condition 1 defines \( X' \) to be a set of non-verdict states of the form \((X, \sigma)\) such that \( \sigma \) is a suspension trace of the already tested product \( \Delta_\lambda(s) \) and the predicate \( bt(X, \sigma) \) holds; whereas, \( X'' \) is the set of non-verdict states reachable from a state in \( X' \) by a trace that is not a suspension trace of the tested product \( \Delta_\lambda(s) \). Condition 2 and 3 are self-explanatory.

As an example, the spinal test suite generated from the test suite in Figure 3 is partially drawn in Figure 4.

![Figure 4: Spinal test suite of the cruise controller](image)

4 Exhaustiveness of Spinal Test Suites

The spinal test suite \( S(\varphi, \lambda) \) contains the spines of those paths from the test suite \( T(s, \varphi) \) that lead to new behavior w.r.t. to the already-tested product \( \lambda \). Next, we show that the spinal test suite \( S(\varphi, \lambda) \) is not necessarily exhaustive for an arbitrary implementation under test, i.e., it may have strictly less testing power than the test suite \( T(s, \varphi) \). We show this through the following example.

**Example 5.** Consider an implementation of a cruise controller with a collision avoidance feature modeled as the IOFTS depicted in Figure 5. Clearly, this implementation is a faulty one as the action ‘rgl’ must be prohibited after detecting an obstacle, i.e., after executing the transition labeled ‘det’.

As soon as we place the test suite (Figure 3) in parallel \(||\) with the above-given implementation, we observe that the following synchronous interactions emerge: on.off.on.det.rgl, which lead to the fail

![Figure 5: A faulty implementation of the cruise controller with control avoidance.](image)
verdict state. However, note that the aforementioned fault in the implementation cannot be detected while interacting with the spinal test suite of Figure 4, because there are no transitions labeled with off in the spinal test suite. Thus, a spinal test suite \( \mathcal{T}(\varphi, \lambda) \) has strictly less testing power than the test suite \( \mathcal{T}(s, \phi) \).

Next, we explore when a spinal test suite \( \mathcal{T}(\varphi, \lambda) \) (where \( \lambda \models \varphi \)) together with a concrete test suite \( \mathcal{T}(s, \lambda) \) have the same testing power as the abstract test suite \( \mathcal{T}(s, \phi) \).

**Definition 8.** Let \( \lambda \models \varphi \). A feature specification \( \Delta_\varphi(s') \) is orthogonal w.r.t \( \Delta_\varphi(s) \) and the product \( \lambda \) iff

\[
\forall s_1, \sigma', a, \sigma'' \left( \text{new}_{s, \lambda}(\sigma', a) \land \Delta_\varphi(s') \xrightarrow{a\sigma''} \Delta_\varphi(s_1) \right) \Rightarrow \exists s_2, \sigma \Delta_\varphi(s') \xrightarrow{\sigma a\sigma''} \Delta_\varphi(s_2) \land \sigma \uparrow \sigma'.
\]

**Example 6.** Recall the feature specification \( \Delta_\varphi(s_0) \) and the product \( \lambda \) (which omits the control avoidance feature) from Example 4. Note that the implementation given in Figure 5 is not orthogonal w.r.t the feature specification \( \Delta_\varphi(s_0) \) and the product \( \lambda \) because the underlined subsequence in “on off on det rgl” cannot be extended with the spine sequence on.

In the remainder, we prove the main result (Theorem 1) of this section that an orthogonal implementation passes the test suite \( \mathcal{T}(s, \phi) \) whenever it passes the concrete test suite \( \mathcal{T}(s, \lambda) \) and the spinal test suite \( \mathcal{T}(\varphi, \lambda) \).

**Lemma 1.** Let \( X_0 \) be the initial state of a test suite \( \mathcal{T}(s, \phi) \) and let \( \lambda \) be a product with \( \lambda \models \varphi \). If \( X_0 \xrightarrow{\sigma'} \text{fail}, \text{new}_{s, \lambda}(\sigma', a) \), and \( \sigma \uparrow \sigma' \) then \( X_0 \xrightarrow{\sigma a\sigma''} \text{fail} \).

**Proof sketch.** Let us first decompose the sequence of transitions \( X_0 \xrightarrow{\sigma'} \text{fail} \) as \( X_0 \xrightarrow{\sigma'} (X, \sigma') \xrightarrow{\sigma''} \text{fail} \), for some \( X \). Then by definition of a spine path we get \( X_0 \xrightarrow{\sigma} (X, \sigma) \). Next, it is straightforward to show by induction on \( \sigma'' \) that \( (X, \sigma) \xrightarrow{a\sigma''} \text{fail} \), whenever \( (X, \sigma') \xrightarrow{a\sigma''} \text{fail} \) and \( \text{new}_{s, \lambda}(\sigma', a) \).

**Theorem 1.** Let \( \Delta_\varphi(s') \) be orthogonal w.r.t. to \( \Delta_\varphi(s) \) and \( \lambda \). If \( \Delta_\varphi(s') \) pass the test suites \( \mathcal{T}(s, \lambda) \) and \( \mathcal{T}(\varphi, \lambda) \), then \( \Delta_\varphi(s') \) passes the test suite \( \mathcal{T}(s, \phi) \).

**Proof.** Let \( X_0 \) be the initial state of the test suite \( \mathcal{T}(s, \phi) \). We will prove this theorem by contradiction. Let \( \Delta_\varphi(s') \) pass the test suites \( \mathcal{T}(s, \lambda) \) and \( \mathcal{T}(\varphi, \lambda) \). Suppose \( \Delta_\varphi(s') \) fails in passing the test suite \( \mathcal{T}(s, \phi) \). Then, there exists the following sequences of transitions \( X_0 \xrightarrow{\sigma} \text{fail} \) and \( \Delta_\varphi(s') \xrightarrow{\sigma} \Delta_\varphi(s') \) (for some \( \sigma, s' \)) in the test suite \( \mathcal{T}(s, \phi) \) and the feature specification \( \Delta_\varphi(s') \).

There are two subcases:

1. Either, \( \sigma \in \text{Straces}(\Delta_\lambda(s)) \). Then, the feature specification \( \Delta_\varphi(s') \) fails to pass the test suite \( \mathcal{T}(s, \phi) \). Hence, a contradiction.

2. Or, \( \sigma \notin \text{Straces}(\Delta_\lambda(s)) \). Then, the sequence of transitions \( X_0 \xrightarrow{\sigma} \text{fail} \) can be decomposed in the following way: \( X_0 \xrightarrow{\sigma_1 a \sigma_2} \text{fail} \) with \( \sigma = \sigma_1 a \sigma_2 \) and \( \text{new}_{s, \lambda}(\sigma_1, a) \).

Since the feature specification \( \Delta_\varphi(s') \) is orthogonal w.r.t. \( \Delta_\varphi(s) \) and \( \lambda \), we have

\[
\exists s_2, \sigma_1 \Delta_\varphi(s') \xrightarrow{\sigma_1 a \sigma_2} \Delta_\varphi(s'_2) \land \sigma_1 \uparrow \sigma_1.
\]

Then, by applying Lemma 1 we get the following path in the spinal test suite: \( X_0 \xrightarrow{\sigma_1 a \sigma_2} \text{fail} \). Thus, \( \Delta_\varphi(s') \) fails to pass the spinal test suite \( \mathcal{T}(\varphi, \lambda) \); hence, a contradiction. \( \square \)
5 Related work

Various attempts have been made regarding formal and informal modeling of SPLs, on which [20, 6, 21, 9, 23] provide comprehensive surveys. By and large, the literature can be classified into two categories: structural modeling and behavioral modeling techniques.

Structural models specify variability in terms of presence and absence of features (assets, artifacts) in various products and their mutual inter-relations. Behavioral models, however, concern the working of features and their possible interactions, mostly based on some form of finite state machines or labeled transition systems. The main focus in behavioral modeling of SPLs (cf. [2, 1, 7, 8, 11, 12, 16]) has been on formal specification of SPLs and adaptation of formal verification (mostly model checking) techniques to this new setting.

In addition, several testing techniques have been adapted to SPLs, of which [19, 14, 18, 10] provide recent overviews. Hitherto, most fundamental approaches to formal conformance testing [4] have not been adapted sufficiently to the SPL setting. The only exception that we are aware of is [17], which presents an LTS-based incremental derivation of test suites by applying principles of regression testing and delta-oriented modeling [5].

Although our work is based on input-output conformance testing, we envisage that the ideas pursued in this paper can be adapted to other fundamental theories of conformance testing, e.g., those based on finite state machines [4, 25].

6 Conclusions

In this paper, we introduced the notion of spinal test suites, which can be used in order to incrementally test different products of a software product line. A spinal test suite only tests the behavior induced by the “new” features and dispenses with re-testing the already-tested behavior, unless this is necessary in order to reach the behavior of the new features.

As future work, we intend to exploit this notion and establish a methodology of testing software product lines, by automatically detecting the optimal order of testing products, which leads to a minimal size of residual test suites (with respect to a given notion of model coverage). In order to effectively use the notion of spinal test suites, we would like to define syntactic criteria that guarantee orthogonality of features.

Acknowledgments

We thank the anonymous reviewers of MBT 2014 for their useful feedback.

References


Spinal Test Suites for Software Product Lines


Generating Complete and Finite Test Suite for ioco: Is It Possible?

Adenilso Simao  
São Paulo University  
São Carlos, São Paulo, Brazil  
adenilso@icmc.usp.br

Alexandre Petrenko  
Centre de recherche informatique de Montreal (CRIM)  
Montreal, Quebec, Canada  
petrenko@crim.ca

Testing from Input/Output Transition Systems has been intensely investigated. The conformance between the implementation and the specification is often determined by the so-called ioco-relation. However, generating tests for ioco is usually hindered by the problem of conflicts between inputs and outputs. Moreover, the generation is mainly based on nondeterministic methods, which may deliver complete test suites but require an unbounded number of executions. In this paper, we investigate whether it is possible to construct a finite test suite which is complete in a predefined fault domain for the classical ioco relation even in the presence of input/output conflicts. We demonstrate that it is possible under certain assumptions about the specification and implementation, by proposing a method for complete test generation, based on a traditional method developed for FSM.

1 Introduction

Testing from Input/Output Transition System (IOTS) has received great attention from academy and industry alike. The main research goal is to devise a theoretically sound testing framework when the behavior of an Implementation Under Test (IUT) is specified as the IOTS model. It is assumed that the tester controls when inputs are applied, while the IUT autonomously controls when, and if, outputs are produced. The IUT’s autonomy causes issues in testing. Simply stated, the interaction between the IUT and the tester should be assumed to be asynchronous, since otherwise the tester should have the ability to block the IUT when the latter is ready to produce output but the former has input to be sent. Most approaches based on the so-called ioco conformance relation do not offer sound solutions to the problem of conflicts between inputs and outputs. In particular, the proposal [15] for input-enabled testers addressing the conflicts lead to uncontrollable tests, while it is widely agreed that only controllable tests, which avoid any choice between inputs or between input and output, should be used. The approaches for test purpose driven test generation from the IOTS implemented in tools such as TGV [8] and TorX [16], as well as in Uppaal Tron which also accepts the IOTS, face the same problem of treating input/output conflicts.

These issues have drawn significant attention of the testing community, e.g., [1, 6, 7, 11], and have been dealt with by allowing implicitly or explicitly the presence of channels, e.g., FIFO queues, between the IUT and tester [6, 7, 17]. However, queues impose a hard burden on the tester, since the communication is now distorted by possible delay in the transmission of messages via queues. In the extreme case, queues render some important testing problems undecidable [4, 5]. The issue is caused by the conflict between input and output enabled in the same state; while the IUT should be ready to receive input, it may choose to produce an output, blocking or ignoring incoming input. It has been shown that when all the states have either inputs or outputs, but not both, in the so-called Mealy IOTS, such problems do not arise [12].
Apart for the problem of input/output conflicts, the question of generating complete and finite test suite from IOTS w.r.t. the $ioco$ relation remains open. The test generation method which is most referred in the literature relies on non-deterministic choice between: (1) stopping testing; (2) applying a randomly chosen input; or (3) checking for outputs [14]. The problem with this approach is that, although completeness is guaranteed in some theoretical sense, the practical application of this method is problematic. It requires that the process be repeated an undetermined number of times, since there is no indication of when the completeness has been achieved and thus the process can stop.

On the other hand, generation methods from Finite State Machines (FSM) approach the problem of test completeness by explicitly stating a set of faulty (mutant) FSMs, called a fault domain, which model potential faults of the IUT; then, a test suite is generated that targets each faulty FSM. Its completeness implies that each IUT possessing the modelled faults will be detected by the test suite. The existing methods for complete test generation are applicable not only to minimal deterministic machines, as the early methods [2, 3, 18], but also to nondeterministic FSMs [10]. This motivated a previous attempt to rephrase FSM methods for checking experiments to the IOTS model [13]. In particular, an analogue of the Harmonized State Identifier Method (HSI-method) was elaborated there for the trace equivalence relation between the specification and implementation IOTSs. The input/output conflicts were addressed by assuming that the tester detecting (using some means) them will just try to repeatedly re-execute the expected trace to verify if it can be generated by the IUT.

In this paper, we investigate whether it is possible to construct a finite test suite for a given IOTS specification which is complete in a predefined fault domain for the classical $ioco$ relation even in the presence of input/output conflicts. Our solution to the latter is based on the assumption that any IUT in a fault domain resolves each such conflict in favor of inputs; that is, we assume that the IUT is eager to process inputs and, whenever it is in a state where it can either receive an input or produce an output, it will produce an output only if no input is available. We demonstrate this by elaborating a test generation method inspired by the HSI method [19], generalizing and adapting its concepts to the realm of IOTS. We illustrate the method with a running example.

This remainder of this paper is organized as follows. In Section 2, we introduce the main concepts of IOTS and test cases. In Section 3, we present the generation method, and demonstrate that the obtained test suite is a complete for a given fault domain. Finally, in Section 4, we conclude the paper and point to future work.

### 2 Input/output transition system and test cases

#### 2.1 Input/output transition system and related definitions

We use input/output transition systems (IOTS, a.k.a. input/output automata [9]) for modelling systems. Formally, an IOTS $S$ is a quintuple $(S, s_0, I, O, h_S)$, where $S$ is a finite set of states and $s_0 \in S$, is the initial state, $I$ and $O$ are disjoint sets of input and output actions, respectively, and $h_S \subseteq S \times (I \cup O) \times S$ is the transition relation. $S$ is deterministic if $h_S$ is a function on a subset of $S \times (I \cup O)$, i.e., if $(s, x, s') \in h_S$ and $(s, x, s'') \in h_S$, then $s' = s''$. While we shall consider only deterministic IOTSs, they may have output-nondeterminism, i.e., have several outputs enabled in a state.

For IOTS $S$, let $init_S(s)$ denote the set of actions enabled at state $s$, i.e., $init_S(s) = \{x \in I \cup O \mid \exists s' \in S, (s, x, s') \in h_S\}$; let $inps(s)$ and $outs(s)$ denote the set of inputs and outputs, respectively, enabled at state $s$. Thus, $inps(s) = init_S(s) \cap I$; $outs(s) = init_S(s) \cap O$. We omit the subscript if it is clear which IOTS is considered.
A state $s$ is a sink state if $\text{init}(s) = \emptyset$; $s$ is an input state if $\text{inp}(s) \neq \emptyset$. We denote the set of input states by $S_{\text{in}}$. An input state $s$ is stable (quiescent) if $\text{init}(s) \subseteq I$. An input state $s$ is a quasi-stable state if $\text{out}(s) \neq \emptyset$. In a quasi-stable state, there is an input/output conflict (note that the IOTS itself does not provide any mechanism for resolving such conflicts). A state is an output state if it is neither sink nor input state. Figure 1 shows an example of an IOTS, where $I = \{a, b\}$ and $O = \{0, 1\}$. Input states are numbered; states 1 and 4 are stable, whereas states 2 and 3 are quasi-stable.

For IOTS $S$, a path from state $s_1$ to state $s_{n+1}$ is a sequence of transitions $p = (s_1, a_1, s_2)(s_2, a_2, s_3) \ldots (s_n, a_n, s_{n+1})$, where $(s_i, a_i, s_{i+1}) \in h_S$ for $i = 1, \ldots, n$. Let $\varepsilon$ denote the empty sequence of actions. We say that $s_{n+1}$ is reachable from $s_1$. IOTS $S$ is initially-connected if each state is reachable from the initial state. A sequence $u \in (I \cup O)^*$ is called a trace of $S$ from state $s_1 \in S$ if there exists path $(s_1, a_1, s_2)(s_2, a_2, s_3) \ldots (s_n, a_n, s_{n+1})$, such that $u = a_1 \ldots a_n$. We use the usual operator after to denote the state reached after the sequence of actions (we consider only deterministic IOTS), i.e., $s_1$-after-$u = s_{n+1}$; if $u$ is not a trace of $s_1$, then $s_1$-after-$u = \emptyset$. Let also $\text{Tr}(T)$ denote the set of traces from states in $T \subseteq S$. For simplicity, we denote $\text{Tr}\{s\}$ as $\text{Tr}(s)$ and use $\text{Tr}(S)$ to denote $\text{Tr}(s_0)$. A trace $u$ of IOTS $S$ is completed, if $s_0$-after-$u$ is a sink state. A trace $u$ of IOTS $S$ is a bridge trace from input state $s$, if $s$-after-$u \in S_{\text{in}}$ and for each proper prefix $w$ of $u$, $s$-after-$w \notin S_{\text{in}}$.

Given an IOTS $S = (S, s_0, I, O, h_S)$ and a state $s \in S$, let $S/s$ denote the IOTS that differs from $S$ in the initial state changed to $s$, removing states and transitions which are unreachable from $s$.

We use a designated symbol $\delta$ to indicate quiescence in $S$, that is, the absence of outputs. Quiescence can be encoded by adding self-looping $\delta$ transitions to the stable states; the resulting IOTS has the output action set $O \cup \{\delta\}$. Traces of this IOTS which end with $\delta$ are quiescent traces and traces containing $\delta$ are suspension traces. In the rest of the paper, we assume that $\text{Tr}(S)$ includes all kinds of traces.

An IOTS $T = (T, t_0, I, O, h_T)$ is a submachine of the IOTS $S = (S, s_0, I, O, h_S)$, if $T \subseteq S$ and $h_T \subseteq h_S$. A state $s \in T$ of a submachine $T$ of $S$ is output-preserving if for each $x \in O$ such that $(s, x, s') \in h_S$, we
have that \((s,x,s') \in h_T\). The submachine \(T\) of \(S\) is \textit{output-preserving} if each state which is not a sink state is output-preserving. The submachine is \textit{trivial} if \(T\) is a singleton and \(h_T = \emptyset\).

The IOTS \(S\) is \textit{progressive} if it has no sink state and each cycle contains a transition labeled with input, i.e., there is no output divergence. The IOTS \(S\) is \textit{input-complete} if all inputs are enabled in input states, i.e., \(\text{inp}(s) \neq \emptyset\) implies that \(\text{inp}(s) = I\), for each state \(s\). The IOTS \(S\) is \textit{single-input} if \(|\text{inp}(s)| = 1\), for each input state \(s\); it is \textit{output-preserving} if \(\text{out}(s) = O\), for each output state \(s\).

In this paper, we assume that specifications and implementations are input-complete progressive deterministic initially-connected IOTS; we let \(\text{IOTS}(I,O)\) denote the set of such IOTSs with input set \(I\) and output set \(O\).

To characterize the common behavior of two IOTSs in \(\text{IOTS}(I,O)\) we use the intersection operation. The intersection \(\mathcal{S} \cap \mathcal{P}\) of IOTSs \(\mathcal{S} = (S,s_0,I,O,h_S)\) and \(\mathcal{P} = (P,p_0,I,O,h_P)\) is an IOTS \((Q,q_0,I,O,h_{\mathcal{S} \cap \mathcal{P}})\) with the state set \(Q \subseteq S \times P\), the initial state \(q_0 = (s_0,p_0)\), and the transition relation \(h_{\mathcal{S} \cap \mathcal{P}}\), such that \(Q\) is the smallest state set obtained by using the rule \(((s,p),x,(s',p')) \in h_{\mathcal{S} \cap \mathcal{P}} \iff (s,x,s') \in h_S \text{ and } (p,x,p') \in h_P\). The intersection \(\mathcal{S} \cap \mathcal{P}\) preserves only common traces of both machines; in other words, for each state \((s,p)\) of \(\mathcal{S} \cap \mathcal{P}\) we have \(\text{Tr}((s,p)) = \text{Tr}(s) \cap \text{Tr}(p)\); moreover, \(\text{out}((s,p)) = \text{out}(s) \cap \text{out}(p)\). Thus, \(\text{Tr}(\mathcal{S} \cap \mathcal{P}) = \text{Tr}(\mathcal{S}) \cap \text{Tr}(\mathcal{P})\).

Given two IOTSs \(\mathcal{S}\) and \(\mathcal{T}\), such that \(\mathcal{S}\) has at least one sink state \(s \in S\), the IOTS obtained by merging the initial state of \(\mathcal{T}\) with a sink state \(s\) is called the \textit{chaining} of \(\mathcal{S}\) and \(\mathcal{T}\) in the sink state \(s\), denoted \(\mathcal{S} @ \mathcal{T}\).

For conformance testing, we consider a usual \text{iso}\co relation.

\textbf{Definition 1} \hspace{1em} \text{Given two IOTSs } \mathcal{S}, \mathcal{P} \in \text{IOTS}(I,O), \mathcal{S} = (S,s_0,I,O,h_S) \text{ and } \mathcal{P} = (P,p_0,I,O,h_P) \text{, we write } \mathcal{P} \text{ \textit{iso}\co } \mathcal{S} \text{ if for each trace } \alpha \in \text{Tr}(\mathcal{S}) \text{, we have that out}((\mathcal{P} \text{-after-} \alpha)) \subseteq \text{out}(\mathcal{S} \text{-after-} \alpha). \text{ If } \mathcal{P} \text{ \textit{iso}\co } \mathcal{S} \text{ then we say that state } p_0 \text{ is a reduction of state } s_0. \text{ The reduction relation between states is also defined for states of the same IOTS } \mathcal{S} \in \text{IOTS}(I,O), \text{ namely, } s_1 \text{ is a reduction of } s_2, \text{ if } \mathcal{S} @ s_1 \text{ \textit{iso}\co } \mathcal{S} @ s_2.\]

We write \(\mathcal{P} \text{ \textit{iso}\co } \mathcal{S}\), if not \(\mathcal{P} \text{ \textit{iso}\co } \mathcal{S}\). We notice that if the specification IOTS \(\mathcal{S}\) contains some state that is a reduction of another state then there exist an implementation \(\mathcal{P} \in \text{IOTS}(I,O)\) and state \(p \in P\), that is a reduction of both states of \(\mathcal{S}\). Intuitively, the two states are “merged” into a single state in the implementation. As a result, a conforming implementation may have fewer states than its specification. This observation motivates the following definitions and statements.

\textbf{Definition 2} \hspace{1em} \text{Two states of } \mathcal{S} \in \text{IOTS}(I,O) \hspace{1em} \text{are compatible, if there exists a state of an IOTS } \mathcal{P} \in \text{IOTS}(I,O) \text{ that is a reduction of both states; otherwise, i.e., if for any } \mathcal{P} \in \text{IOTS}(I,O), \text{ no state of } \mathcal{P} \text{ is a reduction of both states, they are distinguishable.} \]

According to this definition, compatible states can be “merged” in an implementation IOTS into a single state and it can still be a reduction of the specification IOTS, however, any reduction of the specification IOTS cannot have a state that is a reduction of distinguishable states.

The compatibility of states can be easily determined by the intersection of IOTSs, a simple and inexpensive operation. By definition, if two states of a given IOTS are compatible, there exists a state of some input-complete, progressive IOTS which is a reduction of both states. Such a state is the initial state of the intersection of two instances of a given machine initialized in different states, since the intersection represents all the common traces of the two states. On the other hand, if the two states are distinguishable, the intersection is not a progressive IOTS. This fact is stated in the following lemma.

\textbf{Lemma 1} \hspace{1em} \text{Two states } s_1, s_2 \in S \text{ of } \mathcal{S} = (S,s_0,I,O,h_S), \mathcal{S} \in \text{IOTS}(I,O) \text{ are compatible if and only if } \mathcal{S} / s_1 \cap \mathcal{S} / s_2 \in \text{IOTS}(I,O).\]
Proof. Suppose that $s_1$ and $s_2$ are compatible. We show that $S/s_1 \cap S/s_2 \in IOTS(I,O)$, that is, $S/s_1 \cap S/s_2$ is input-complete, progressive, deterministic and initially-connected. Let $\alpha \in Tr(S/s_1 \cap S/s_2)$. Thus, $\alpha \in Tr(S/s_1) \cap Tr(S/s_2)$. We have that $s'_1 = S/s_1$-after-$\alpha$ and $s'_2 = S/s_2$-after-$\alpha$ are also compatible. Hence, by Definition 1, there exists a state $p \in IOTS(I,O)$, with $P = (p, p_0, I, O, h_0)$ that is a reduction of $s'_1$ and $s'_2$. It holds that $out(p) \subseteq out(s'_1)$ and $out(p) \subseteq out(s'_2)$. As $P$ is progressive, we have that $init(p) \neq \emptyset$, and thus there exists $x \in out(p)$; hence, $x \in init(s'_2) \cap init(s'_2)$. It follows that $(S/s_1 \cap S/s_2)$-after-$\alpha$ is not a sink state, since it is followed by $x$, at least. Thus, $S/s_1 \cap S/s_2$ has no sink state. If $x$ is an input, then $I \subseteq init(s'_1)$ and $I \subseteq init(s'_2)$, since $S$ is input-complete. Therefore, $I \subseteq init((S/s_1 \cap S/s_2)$-after-$\alpha)$, and $S/s_1 \cap S/s_2$ is input-complete. As $S$ is progressive, it does not have cycles with transitions labeled only with outputs. Hence, neither $S/s_1 \cap S/s_2$ has such cycles, i.e., $S/s_1 \cap S/s_2$ is also progressive. As $S$ is deterministic and initially-connected, so are $S/s_1, S/s_2$ and, consequently, $S/s_1 \cap S/s_2$. It follows then that $S/s_1 \cap S/s_2 \in IOTS(I,O)$.

Suppose now that the intersection $S/s_1 \cap S/s_2 \in IOTS(I,O)$, i.e., it is input-complete, progressive, deterministic and initially-connected. We show that $s_1$ and $s_2$ are compatible, demonstrating that the initial state of $S/s_1 \cap S/s_2$ is a reduction of $s_1$ and $s_2$. For each trace $\alpha \in Tr(S/s_1 \cap S/s_2)$, we have that $init((S/s_1 \cap S/s_2)$-after-$\alpha) \subseteq init((S/s_1$-after-$\alpha) = init(s_1$-after-$\alpha)$; thus, $init((S/s_1 \cap S/s_2)$-after-$\alpha) \cap O = out((S/s_1 \cap S/s_2)$-after-$\alpha) \subseteq init((S/s_1$-after-$\alpha) \cap O = out((S/s_1$-after-$\alpha) = out((S/s_1$-after-$\alpha)$). Therefore, the initial state of $S/s_1 \cap S/s_2$ is a reduction of $s_1$. Analogously, $S/s_1 \cap S/s_2$ is a reduction of $s_2$ and the result thus follows.

Corollary 1 States $s_1$ and $s_2$ of $S$ are distinguishable if and only if $S/s_1 \cap S/s_2 \notin IOTS(I,O)$, i.e., the IOTS $S/s_1 \cap S/s_2$ has a sink state.

An IOTS in IOTS(I,O) is input-state-minimal if every two input states are distinguishable. In the following, we assume that IOTSs which are not input-state-minimal are excluded from IOTS(I,O).

The next lemma states when one state of an IOTS is a reduction of another. The outputs enabled in each state reached in the intersection IOTS, initialized with the respective states, are exactly the outputs enabled in one of the states.

Lemma 2 Given two states $s_1, s_2 \in S$ of $S = (S, s_0, I, O, h_S)$, $s_1$ is a reduction of $s_2$ if and only if $out((s, s')) = out(s)$ for each state $(s, s')$ of $S/s_1 \cap S/s_2$.

Proof. Assume that $s_1$ is a reduction of $s_2$; thus, $S/s_1$ ioco $S/s_2$. We have that for each trace $\alpha \in Tr(S/s_2)$, $out(s_1$-after-$\alpha) \subseteq out(s_2$-after-$\alpha)$. Let $(s, s')$ be a state of $S/s_1 \cap S/s_2$. Thus, there exists a trace $\beta \in Tr(S/s_1 \cap S/s_2)$, such that $(S/s_1 \cap S/s_2$)-after-$\beta = (s, s')$ and, therefore, $S/s_1$-after-$\beta = s$ and $S/s_2$-after-$\beta = s'$. It holds that $\beta \in Tr(S/s_2)$ and $out(s_1$-after-$\beta) \subseteq out(s_2$-after-$\beta)$; thus, $out(s) \subseteq out(s')$. We have that $out((s, s')) = out(s) \cap out(s')$. The result then follows, since $out(s) \subseteq out(s')$ and $out((s, s')) = out(s) \cap out(s')$ implies that $out((s, s')) = out(s)$. Assume now that $out((s, s')) = out(s)$ for each state $(s, s')$ of $S/s_1 \cap S/s_2$. Let $\alpha \in Tr(S/s_2)$. We have that $\alpha \in Tr(S/s_1)$ if and only if $\alpha \in Tr(S/s_1 \cap S/s_2)$. If $\alpha \notin Tr(S/s_1)$, then $out((S/s_1$-after-$\alpha) = \emptyset$ and the result follows, since $out((S/s_1$-after-$\alpha) \subseteq out((S/s_2$-after-$\alpha)$. If $\alpha \in Tr(S/s_1)$, let $(s, s') = s_1$-after-$\alpha \cap S/s_2$-after-$\alpha$; thus, $s = s_1$-after-$\alpha$ and $s' = s_2$-after-$\alpha$. We have that $out((S/s_1$-after-$\alpha \cap S/s_2$-after-$\alpha) = out((S/s_1$-after-$\alpha)$. Let $x \in out((S/s_1$-after-$\alpha)$. As $x \in out((S/s_1$-after-$\alpha \cap S/s_2$-after-$\alpha) = out((S/s_1$-after-$\alpha) \cap out((S/s_2$-after-$\alpha)$, it holds that $x \in out((S/s_2$-after-$\alpha)$. The result then follows, since $out((S/s_1$-after-$\alpha) \subseteq out((S/s_2$-after-$\alpha)$, implying that $S/s_1$ ioco $S/s_2$, i.e., $s_1$ is a reduction of $s_2$.\)
2.2 Test definitions and problem statement

To simplify the discussion, we refer to inputs and outputs always taking the view of the implementation, IUT; thus, we say, for instance, that the tester sends an input to the IUT and receives outputs from it, and define test cases accordingly preserving the input and output sets of the specification $IOTS = (S, s_0, I, O, h_S)$. Recall that $\delta$ is included into $O$; in particular, the output $\delta$ of a test case is interpreted as the fact that the tester executing the test case detects quiescence of the IUT.

**Definition 3** A test case over input set $I$ and output set $O$ is an acyclic single-input output-complete $IOTS \ U = (U, u_0, I, O, h_U)$, where $U$ has a designated sink state fail. A test case is controllable if it has no quasi-stable states, otherwise it is uncontrollable. A test suite is a finite set of test cases.

Let $Tr_{fail}(U)$ be the traces which lead to the sink state $fail$, i.e., $Tr_{fail}(U) = \{ \alpha \in Tr(U) \mid U-\text{after-} \alpha = fail \}$. Let $Tr_{pass}(U)$ be the traces which do not lead to $fail$, i.e., $Tr_{pass}(U) = Tr(U)\setminus Tr_{fail}(U)$.

**Definition 4** Given the specification $IOTS S$, a test case $U = (U, u_0, I, O, h_U)$, and an implementation $IOTS B \in IOTS(I, O)$,

- $B$ passes the test case $U$, if the intersection $B \cap U$ has no state, where the test $U$ is in the state fail.
- $B$ fails $U$, if the intersection $B \cap U$ has a state, where the test $U$ is in the state fail.

A test suite $T$ is

- sound for $IOTS S$ in $IOTS(I, O)$, if each $B \in IOTS(I, O)$, such that $B$ ioco $S$, passes each test in $T$.
- exhaustive for $IOTS S$ in $IOTS(I, O)$, if each $IOTS B \in IOTS(I, O)$, such that $B$ ioco $S$, fails some test in $T$.
- complete for $IOTS S$ in $IOTS(I, O)$ w.r.t. the ioco relation, if $T$ is sound and exhaustive for $S$ in $IOTS(I, O)$.

Notice that $B$ passes the test case $U$, if and only if $Tr(B) \cap Tr_{fail}(U) = \emptyset$ and $Tr_{pass}(U) \subseteq Tr(B)$.

The problem of complete test suite generation for a given $IOTS$ was addressed in [14, 15]. To generate such a test suite a simple algorithm is suggested, which, however, should be executed an indeterminate number of times to achieve the test completeness w.r.t. the ioco relation. In the first work [14], only controllable test cases are generated; the problem with that solution is that the tester must be able to somehow preempt any output each time a test case prescribes sending some input to the IUT. In the second work [15], “the most important technical change with respect to [14] is the input enabledness of test cases, which was inspired by [11]”. In terms of our definitions, test cases are uncontrollable; they contain quasi-stable states, where both inputs and outputs are enabled. The intention behind this is to address input/output conflict present in the specification $IOTS$, since the specification itself provides no clue how an implementation resolves input/output conflict. The behavior of the tester executing uncontrollable test cases may become nondeterministic (the tester has to execute one of the two mutually exclusive actions) and the test results may not always be reproducible. The approaches to generation of uncontrollable tests that tolerate input/output conflicts based on the use of queues are elaborated in several work [4, 6, 7, 11, 12, 17]. The problem is that one needs to know the size of queues to obtain a finite complete test suite.

In this paper, we demonstrate, first, that controllable tests that tolerate input/output conflicts can be constructed without knowing the size of queues, and second, that it is possible to obtain in a systematic...
way a finite set of controllable test cases which is a complete test suite in a finite set of IOTSs. The key assumption we make about the implementation IOTSs in the fault domain is that each implementation when it is a quasi-stable state with the input/output conflict, it does not produce any output if its input queue contains an input. We call such implementations input-eager. A subset of \( IOTS(1, O) \) that contains input-eager IOTSs is denoted \( IEIOTS(1, O) \). Finiteness of complete test suites results from further constraining this set by the number of its input states, as we demonstrate later.

Testing any input-eager IOTS allows one to use two controllable test cases dealing with input/output conflict; in a quasi-stable state one test case does not send any input and only observes output sequence concluded by quiescence and another one just sends input. In the latter case, the tester does not need to preempt IUT outputs, as an input-eager IOTS will not produce them since the input queue is not empty and contains the input from the tester.

### 3 Generating complete test suites for IOTS

In this section, we investigate whether a classical method for constructing a complete test suite for the FSM model can be reworked to achieve the same result for the IOTS model even with input/output conflicts, namely a test suite with controllable test cases complete in a finite fault domain, without transforming IOTS into Mealy machine. To demonstrate that it is in fact possible, we develop here a counter-part of the HSI-method [19] for the simplest case, when the FSM is completely specified, minimal, and the fault domain contains FSMs with the number of states not exceeding that of the specification machine.

The HSI-method for FSMs uses sets of distinguishing input sequences, so-called harmonized state identifiers, one per state, such that any two identifiers share an input sequence which distinguishes the two states. These input sequences are appended to state and transition covers in order to check that every state of the implementation corresponds to some state of the specification and every transition of the implementation corresponds to a transition of the specification.

Accordingly, we need first to define state and transition covers, as well as harmonized state identifiers for a given IOTS.

#### 3.1 State and transition covers for IOTS

We first turn our attention to the notion of state cover, needed in tests to eventually establish a mapping from states of the specification to states of the IUT. We focus only on input states of the specification IOTS. First, to check the IUTs reaction to some input it is in fact sufficient to apply the input to a given input state, observe an output sequence, and if it is correct then check whether a proper input state is reached. Output state identification can thus be avoided. However, even considering only input states, some input state of the specification may not be mapped to any state of the IUT even if the latter is a reduction of the specification. Therefore, we should define a state cover targeting only those input states of the specification which have a corresponding state in any \( \text{ioco} \)-conforming implementation.

**Definition 5** Given an initially connected IOTS \( S \) and an input state \( s \), \( s \) is certainly reachable (c-reachable), if any \( \mathcal{P} \in IOTS(1, O) \), such that \( \mathcal{P} \text{ioco} \ S \), contains an input state that is a reduction of \( s \).

It turns out that the certainly reachable states can be determined by considering a submachine of \( S \), similarly to the FSM case [10].

**Lemma 3** An input state \( s \) of an IOTS \( S \) is c-reachable if \( S \) contains a single-input acyclic output-preserving submachine of \( S \) which has \( s \) as the only sink state.
Proof. Let \( C_s \) be a single-input acyclic output-preserving submachine of \( S \), which has \( s \) as the sink state. The input state \( s \) is the only sink state in the submachine; hence all its completed traces converge in \( s \). The submachine is output-preserving, which means that for each \( \alpha \in Tr(C_s) \), if \( C_s\)-after-\( \alpha \) \( \neq s \) then \( out(C_s\text{-after-}\alpha) = out(S\text{-after-}\alpha) \). Hence for any IOTS \( P \in IOTS(I,O) \), such that \( P \) \text{ ioco } \( S \), it also holds that \( out(P\text{-after-}\alpha) \subseteq out(S\text{-after-}\alpha) \), thus \( out(P\text{-after-}\alpha) \subseteq out(C_s\text{-after-}\alpha) \). This implies that \( P \) should have at least one of the completed traces of \( C_s \); let \( \beta \) be such a completed trace. It is easy to see that \( P \) \text{ ioco } \( S \) implies that for any \( \gamma \in Tr(P) \), \( P\text{-after-}\gamma \) is a reduction of \( S\text{-after-}\gamma \). Hence in any IOTS \( P \in IOTS(I,O) \), such that \( P \) \text{ ioco } \( S \), the state \( P\text{-after-}\beta \) is a reduction of \( S\text{-after-}\beta \). The result follows, since \( \beta \) is a completed trace of \( C_s \) and, thus, \( S\text{-after-}\beta = s \).

**Definition 6** Given a c-reachable input state \( s \) of an IOTS \( S \), a single-input acyclic output-preserving submachine \( C_s \), which has \( s \) as the only sink state, is a preamble for state \( s \).

Preambles for states can be determined by Algorithm 1, adapted from [10].

**Algorithm 1** for constructing a preamble for a given input state.

**Input:** An IOTS \( S \) and input state \( s \in S \).

**Output:** a preamble if the state \( s \) is c-reachable.

Construct an IOTS \( R = (R, r_0, I, O, h_R) \) as follows

\[
\text{R} := \{s\}; \\
h_R := \emptyset;
\]

While \( s_0 \notin R \) and there exist an input state \( s' \notin R \) and nonempty \( A \subseteq I \), such that for each \( x \in A \), \( (s', x, s'') \in h_B \), and for each trace \( \gamma \in Tr(s'') \), where \( \gamma \in O^* \), there exists a prefix \( \gamma' \) such that \( s''\text{-after-}\gamma' \in R \)

\[
R := R \cup \{s'\} \cup \{s''\text{-after-}\alpha \mid \gamma \in O^*, \gamma \in Tr(s''), \alpha \in \text{pref}(\gamma')\}; \\
h_R := h_R \cup \{(s', x, s'') \in h_B \mid x \in A\} \cup \{(s''\text{-after-}\alpha, o, s'''\text{-after-}\alpha o) \mid \gamma \in O^*, \gamma \in Tr(s''), \alpha o \in \text{pref}(\gamma')\};
\]

End While;

If \( s_0 \notin R \) then return the message “the state \( s \) is not c-reachable” and stop;
Else let \( R = (R, r_0, I, O, h_R) \), where \( r_0 := s_0 \), be the obtained IOTS;
Starting from the initial state, remove in each state all input transitions, but one, to obtain a single-input submachine with the only sink state \( s \);
Delete states which are unreachable from the initial state;
Return the obtained machine as a preamble for the state \( s \) and stop. \( \lozenge \)

A preamble can be used to transfer from the initial state to c-reachable input states. For the initial state itself, the preamble is simply the trivial IOTS, which contains only the initial state. Figures 2.a, 2.b and 3 show the preambles for states 2, 3 and 4, respectively, of the IOTS in Figure 1.

We assume that each input state of the specification IOTS \( S \) is c-reachable and the initial state is a stable state. An input state cover \( Z \) of \( S \) is a set of preambles, one for each input state, i.e., \( Z = \{C_s \mid s \in S_{in}\} \).

In FSM-based testing, a state cover is extended to a transition cover, by adding all inputs to each transfer sequence of the state cover. In an IOTS, an input applied in an input state may be followed by a number of output sequences leading to various stable states, creating quiescent traces of IOTS. The set of all possible quiescent traces created by \( x \in I \) in input state \( s \in S_{in} \) is \( \{x\gamma\delta \in Tr(s) \mid \gamma \in O^*\} \). We use \( Cov(s, x) \), called \((s, x)\)-cover, to refer to an IOTS, such that \( Tr(Cov(s, x)) = \{x\gamma\delta \in Tr(s) \mid \gamma \in O^*\} \) and the set of sink states is \( \{s\text{-after-}x\gamma \mid \gamma \in O^*\} \). For instance, \( Cov(2, a) \) for state 2 and input \( a \) of
Generating Complete and Finite Test Suite for ioco: Is It Possible?

Figure 2: Preambles $\mathcal{C}_2$ and $\mathcal{C}_3$.

Figure 3: Preamble $\mathcal{C}_4$. 
the IOTS in Figure 1 has the trace $a01\delta$, whereas $Cov(1,a)$ has the traces $a01\delta$, $a11\delta$ and $a10\delta$.

A transition cover $V$ of $S$ is the set of preambles of an input state cover chained with $(s,x)$-covers, i.e., $V = \{ \epsilon \circ \alpha \circ Cov(s,x) \mid s \in S_{in}, x \in I \}$. Notice that each bridge trace starting from a quasi-stable state $s \in S_{in}$ is covered by $Cov(s',x)$, for some input state $s'$ and input $x$. More generally, we state the following lemma.

**Lemma 4** Given an IOTS $S \in IOTS(I,O)$ and a bridge trace $\beta$ from an input state $s \in S_{in}$, there exist input state $s' \in S_{in}$ and input $x$, such that $\gamma \beta \gamma \delta \in Tr(Cov(s',x))$, for some traces $\gamma \in Tr(s')$ and $\gamma \in Tr(s' -$after-$\gamma \beta)$.

**Proof.** If $\beta$ starts with an input, then the results follows directly, since with $\gamma$ as the empty sequence $\beta \gamma \delta$ is a quiescent trace starting at state $s$. If $\beta$ starts with an output, then, $\beta \in O^*$ and $s$ is a quasi-stable state. Notice that there exists $\gamma \in O^*$, such that $\beta \gamma \delta \in Tr(s)$, since $S$ is progressive. Moreover, there exist an input state $s'$, a trace $\gamma$ starting with $x$ and followed by outputs, such that $s'$-$after-\gamma = s$. Thus, $\gamma \beta \gamma \delta \in Tr(Cov(s',x)).$

### 3.2 State identifiers for IOTS

The notion of a separator for two states of a given IOTS can be considered as the generalization of the notion of separating sequence used for FSM.

**Definition 7** Given distinguishable states $s_1$ and $s_2$ of an IOTS $S \in IOTS(I,O)$, a single-input acyclic IOTS $R(s_1,s_2) = (R,r_0,I,O,h_R)$ with the sink states $\perp_{s_1}$ and $\perp_{s_2}$ is a separator of states $s_1$ and $s_2$ if the following two conditions hold:

- $r_0 -$after-$\alpha = \perp_{s_1}$ implies $\alpha \in Tr(s_1) \setminus Tr(s_2)$ and $r_0 -$after-$\alpha = \perp_{s_2}$ implies $\alpha \in Tr(s_2) \setminus Tr(s_1)$;
- for each trace $\alpha$ of $R(s_1,s_2)$ and input $x$ defined in $r_0 -$after-$\alpha$, $\text{out}(r_0 -$after-$\alpha x) = \text{out}(s_1 -$after-$\alpha x) \cup \text{out}(s_2 -$after-$\alpha x)$.

The IOTS, obtained by removing from $R(s_1,s_2)$ the sink state $\perp_{s_2}$ and all transitions leading to it, is called a distinguisher of $s_1$ from $s_2$ and is denoted by $W(s_1,s_2)$.

Separator $R(s_1,s_2)$ can be obtained from the intersection $S/s_1 \cap S/s_2 = (Q, (s_1,s_2), I,O,h_S/s_1 \cap S/s_2)$, similar to the case of FSM [10], as follows (Algorithm 2). First we determine the intersection $S/s_1 \cap S/s_2$ and identify the states where the two IOTSs $S/s_1$ and $S/s_2$ disagree on outputs. For each such state, we add transitions leading to sink states $\perp_{s_1}$ and $\perp_{s_2}$. In the final step, we determine a separator as a single-input output-preserving acyclic submachine of the obtained IOTS by removing inputs, as in Algorithm 1.

---

**Algorithm 2** for constructing a separator for two input states.

**Input:** An IOTS $S$ and distinguishable input states $s_1, s_2 \in S_{in}$.

**Output:** a separator $R(s_1,s_2)$.

1. Construct the IOTS $S/s_1 \cap S/s_2 = (Q, (s_1,s_2), I,O,h_S/s_1 \cap S/s_2)$
2. Let $Q_{dis} = \{(s,s') \in Q \mid \text{out}(s) \neq \text{out}(s')\}$
3. $h_{dis} = \{(s,s'), \alpha, \perp_{s_1} \mid (s,s') \in Q_{dis}, \alpha \in \text{out}(s) \setminus \text{out}(s')\} \cup \{((s,s'), \alpha, \perp_{s_2} \mid (s,s') \in Q_{dis}, \alpha \in \text{out}(s') \setminus \text{out}(s)\}
4. $h_{\perp} = h_{S/s_1 \cap S/s_2} \cup h_{dis}$
5. Let $P = (Q \cup \{\perp_{s_1}, \perp_{s_2}\}, (s_1,s_2), I,O,h_{\perp})$
6. Starting from the initial state, remove in each state all input transitions, but one, to obtain a single-input submachine with the only sink states $\perp_{s_1}$ and $\perp_{s_2}$. 

Delete states which are unreachable from the initial state;
Return the obtained machine as a separator for the states $s_1$ and $s_2$, and stop. ◊

Notice that a separator of states $s_1$ and $s_2$ is obviously a separator of $s_2$ and $s_1$, i.e., $\mathcal{R}(s_1, s_2) = \mathcal{R}(s_2, s_1)$, whereas a distinguisher of $s_1$ from $s_2$ is different from a distinguisher of $s_2$ from $s_1$, i.e., $\mathcal{W}(s_1, s_2) \neq \mathcal{W}(s_2, s_1)$. Figure 4 shows a separator $\mathcal{R}(1, 4)$ obtained by Algorithm 2, as well as the corresponding distinguishers $\mathcal{W}(1, 4)$ and $\mathcal{W}(4, 1)$.

We consider only input-state-minimal specification IOTS, so we are interested in distinguishers of only input states. If $s_1$ is a stable state and $s_2$ is a quasi-stable state then the separator $\mathcal{R}(s_1, s_2)$ is simple; it has a transition with $\delta$ leading from the state $(s_1, s_2)$ to state $s_1$ and a transition for each $o \in \text{out}(s_2)$, leading to $s_2$. Thus, a distinguisher of each stable state from any quasi-stable state has a single $\delta$-transition, we call it a quiescence distinguisher of a stable state $s$, denoted $\mathcal{W}^\delta(s)$. It should be included into a stable state identifier of the state $s$.

**Definition 8** A state identifier of input state $s$, denoted $\mathcal{ID}(s)$, is a set of distinguishers $\mathcal{W}(s, s')$ for each input state $s'$ distinguishable from $s$, including $\mathcal{W}^\delta(s)$ if state $s$ is stable. A set of input state identifiers $\{\mathcal{ID}(s) \mid s \in S_{\text{in}}\}$, is harmonized, if for each pair of input states $s_1$ and $s_2$, such that both are either stable or quasi-stable states, there exists a separator $\mathcal{R}(s_1, s_2)$, such that $\mathcal{W}(s_1, s_2) \in \mathcal{ID}(s_1)$ and $\mathcal{W}(s_2, s_1) \in \mathcal{ID}(s_2)$.

For the IOTS in Figure 1, we have that $\mathcal{ID}(1)$ includes $\mathcal{W}^\delta(1)$ as well as $\mathcal{W}(1, 4)$ in Figure 4.
3.3 Complete test suite

Given the specification IOTS $S = (S, s_0, I, O, h_S)$, $S \in IOTS(I, O)$, let $Z$ be an input state cover, $V$ be a transition cover of $S$, and $\{\mathcal{D}(s) \mid s \in S_{in}\}$ be a set of harmonized identifiers for input states. Consider the set of IOTSs obtained by chaining each IOTS from the input state cover and transition cover with a corresponding harmonized state identifier, namely $D = \{\mathcal{D}(s) \mid s \in sink(\mathcal{I}), \mathcal{I} \in (Z \cup V), \mathcal{R} \in \mathcal{D}(s)\}$, where $sink(\mathcal{I})$ is the set of sink states of $\mathcal{I}$. Each IOTS $\mathcal{U} \in D$ is an acyclic single-input IOTS, since it is obtained by chaining IOTSs with these properties. Moreover, it has no quasi-stable states. If the IOTS $\mathcal{U}$ happens to be also output-complete then it satisfies Definition 3 and is already a test case. The IOTSs in this set can easily be completed with the state $fail$ as follows. Given a single-input acyclic IOTS $\mathcal{U} = (U, u_0, I, O, h_U)$, let $TC(\mathcal{U})$ be the IOTS $(T \cup \{fail\}, u_0, I, O, h_U \cup h_f)$, where $h_f = \{(s, o, fail) \mid s \in U, out(s) \neq \emptyset, o \in O \setminus out(s)\}$, which is a test case. Figure 5 shows the example of a test case, obtained by chaining the preamble $\mathcal{C}_2$, $Cov(2, a)$ with the quiescent trace $a01\delta$, and distinguisher $W(1, 4)$. Notice that the quiescence distinguisher $W^{\delta}(4)$ of a stable state 4 is also used to identify this state, since the quiescent trace $a01\delta$ has it as a suffix. The $fail$ state is replicated to reduce the clutter.

Completing each IOTS in the set $D$, we finally obtain a test suite $TS = \{TC(\mathcal{U}) \mid \mathcal{U} \in D\}$. Consider now the subset of $IEIOTS(I, O)$ restricted by the number of input states less or equal to that of the specification IOTS $S$; we denote it by $IEIOTS(I, O, k)$, where $k$ is the number of input states in $S$. We state the main result of the paper.

**Theorem 1** Given an IOTS $S \in IOTS(I, O)$ with $k$ input states, the test suite $TS$ is a complete test suite for $S$ in $IEIOTS(I, O, k)$ w.r.t. $ioco$ relation.

Before proving Theorem 1, we state some auxiliary results.

**Lemma 5** Given two IOTSs $\mathcal{P}$, $S \in IOTS(I, O)$, if $\mathcal{P}$ is an initially connected submachine of $S$ with the same initial state $s_0$, then $\mathcal{P}$ $ioco$ $S$.

**Proof.** Let $\alpha$ be a trace of $S$. We show that $out(\mathcal{P}$-after-$\alpha) \subseteq out(S$-after-$\alpha)$. Let $s = S$-after-$\alpha$. If $s \notin P$, where $P$ is the set of states of $\mathcal{P}$, then $out(\mathcal{P}$-after-$\alpha) = \emptyset$, and the result follows. If $s \in P$, we have that $out(\mathcal{P}$-after-$\alpha) \subseteq out(S$-after-$\alpha)$. As $s = \mathcal{P}$-after-$\alpha$, the result also follows. Thus, $\mathcal{P}$ $ioco$ $S$.
Definition 9  Given two IOTSs \( P, S \in IOTS(I, O) \), \( P = (P, p_0, I, O, h_P) \) and \( S = (S, s_0, I, O, h_S) \), \( P \) is input-state homeomorphic to \( S \), if there exists a bijective map \( \phi \) from \( P_{in} \) to \( S_{in} \) such that for every state \( p \in P_{in} \), each bridge trace \( \gamma \in Tr(p) \), it holds that \( \phi(p) \)-after-\( \gamma = \phi(p) \)-after-\( \gamma \).

\( P \) and \( S \) are input-state isomorphic, if \( P \) is input-state homeomorphic to \( S \) and \( S \) is input-state homeomorphic to \( P \).

Notice that for output-deterministic IOTSs, input-state isomorphic IOTSs are also input-state homeomorphic. An output-nondeterministic IOTS \( S \) that is input-state homeomorphic to \( P \) differs from \( P \) in state names, as well as in the set of bridge traces in some states, since it may have fewer bridge traces, while input-state isomorphic IOTSs differ just in state names.

Corollary 2  Given two IOTSs \( P, S \in IOTS(I, O) \), if \( P \) is input-state homeomorphic to \( S \), then \( P \) is input-state isomorphic to an initially connected submachine of \( S \) with \( k \) input states and the same initial state.

Lemma 6  Given an IOTS \( S \in IOTS(I, O) \), let \( N \in IEIOTS(I, O, k) \) be an IEIOTS which passes \( TS \). Then \( N \) is input-state homeomorphic to \( S \).

Proof. Let \( N \in IEIOTS(I, O, k) \), such that \( N \) passes \( TS \). \( TS \) contains test cases where preambles of an input state cover are chained with harmonized identifiers to the respective states. Thus, for input states \( s \) and \( s' \), \( TS \) contains the test cases \( TC(C_s @ s, W(s, s')) \) and \( TC(C_s @ s, W(s', s)) \). Let \( \alpha \) be a completed trace of \( C_s \) and \( \alpha' \) be a completed trace of \( C_s' \), such that \( \alpha, \alpha' \in Tr(N) \). As \( N \) passes \( TS \), no fail state is reached when the distinguishers \( W(s, s') \) and \( W(s', s) \) are applied after \( \alpha \) and \( \alpha' \), respectively. Since no state can reach sink state in both distinguishers (see Definition 7), we have that the states \( N\)-after-\( \alpha \) and \( N\)-after-\( \alpha' \) are different, i.e., \( N\)-after-\( \alpha \neq N\)-after-\( \alpha' \). These are input states, thus, for each pair of input states of \( N \) there exist a pair of distinct states in \( N \); consequently, \( N \) has at least \( k \) input states. As \( N \in IEIOTS(I, O, k) \), \( N \) has exactly \( k \) input states.

Let \( T \in (Z \cup V) \), \( t \in sink(T) \), \( \alpha \in Tr(T) \cap Tr(N) \), such that \( T\)-after-\( \alpha = t \), \( N\)-after-\( \alpha \in N_{in} \). Similarly, let \( T' \in (Z \cup V) \), \( t' \in sink(T') \), \( \alpha' \in Tr(T') \cap Tr(N) \), such that \( T'\)-after-\( \alpha' = t' \). Notice that \( \alpha \) and \( \alpha' \) are completed traces of IOTSs in the state or transition cover, which are also traces of \( N \). We prove that \( S\)-after-\( \alpha' \) \( \neq S\)-after-\( \alpha \) if and only if \( N\)-after-\( \alpha \neq N\)-after-\( \alpha' \). Suppose first that \( S\)-after-\( \alpha' \) \( \neq S\)-after-\( \alpha \). Thus, \( TS \) contains \( TC(T @ s, W(s, s')) \) and \( TC(T @ s, W(s', s)) \), and as \( N \) passes \( TS \), no fail state is reached when the distinguishers \( W(s, s') \) and \( W(s', s) \) are applied after \( \alpha \) and \( \alpha' \), respectively. Since no state can reach sink state in both distinguishers, we have that \( N\)-after-\( \alpha \) \( \neq N\)-after-\( \alpha' \). Suppose now that \( S\)-after-\( \alpha' \) \( \neq S\)-after-\( \alpha \). We prove by contradiction that \( N\)-after-\( \alpha' \) \( = N\)-after-\( \alpha \). Assume that \( N\)-after-\( \alpha' \) \( \neq N\)-after-\( \alpha \). Thus, let \( s'' \) be an input state, different from \( s = S\)-after-\( \alpha \). Let \( \beta \in Tr(C_s) \), such that \( \beta \in Tr(N) \). As \( TS \) contains \( TC(T @ s, W(s, s'')) \) and \( TC(T @ s, W(s', s)) \), \( N \) passes \( TS \), we have that \( N\)-after-\( \alpha \) \( \neq N\)-after-\( \beta \). Analogously, we can show that we have that \( N\)-after-\( \alpha' \) \( \neq N\)-after-\( \beta \). Thus, \( N\)-after-\( \alpha \) is distinct from \( k - 1 \) distinct input states of \( N \) and \( N\)-after-\( \alpha' \) is also distinct from \( k - 1 \) distinct input states of \( N \). As \( N\)-after-\( \alpha' \) \( \neq N\)-after-\( \alpha \), \( N \) has \( k + 1 \) states, which contradicts the fact that \( N \in IEIOTS(I, O, k) \) and has at most \( k \) input states. Therefore, \( N\)-after-\( \alpha' \) \( = N\)-after-\( \alpha \). Thus, let \( \phi \) be a bijection from the input states \( N_{in} \) of \( N \) to the input states \( S_{in} \) of \( S \), such that for each completed trace \( \chi \) of an IOTS in the state cover \( Z \) or transition cover \( V \), which is also a trace of \( N \), we have that \( \phi(N\)-after-\( \chi \) \( = S\)-after-\( \chi \). Let \( p \) be an input state of \( N \). There exists a completed trace \( \alpha \) of an IOTS in the input state cover \( Z \), such that \( \alpha \) is also a trace of \( N \) and \( N\)-after-\( \alpha = p \). Thus, it holds that \( \phi(N\)-after-\( \alpha \) \( = \phi(p) \) = \( S\)-after-\( \alpha \). Let \( \gamma \in Tr(p) \) be a bridge trace, such that \( \alpha \gamma \) is a completed trace of an IOTS in the transition cover \( V \). Thus, it follows that \( \phi(p)\)-after-\( \gamma = \phi(N\)-after-\( \alpha \)\)-after-\( \gamma = (S\)-after-\( \alpha \)\)-after-\( \gamma = S\)-after-\( \alpha \gamma = \phi(N\)-after-\( \alpha \)\)-after-\( \gamma = \phi(N\)-after-\( \alpha \)\)-after-\( \gamma \).
\( \phi((N\text{-after-}\alpha)\text{-after-}\gamma) = \phi(p\text{-after-}\gamma) \), i.e., \( \phi(p)\text{-after-}\gamma = \phi(p\text{-after-}\gamma) \). Therefore, we have that \( N \) is input-state homeomorphic to \( S \).

We can now prove Theorem 1.

**Proof of Theorem 1.** We first prove that \( TS \) is sound for \( S \) in \( IEIOTS(I,O,k) \). Let \( N \in IEIOTS(I,O,k) \), such that \( N \text{ioco} S \). We have that for each test \( \mathcal{U} \in TS \), \( Tr_{pass}(\mathcal{U}) \subseteq Tr(S) \). Thus, \( Tr_{pass}(\mathcal{U} \cap S) = Tr_{pass}(\mathcal{U}) \cap Tr(S) = Tr_{pass}(\mathcal{U}) \). Since \( N \text{ioco} S \), we have, for each \( \alpha \in Tr(S) \), \( out(N\text{-after-}\alpha) \subseteq out(S\text{-after-}\alpha) \). Let \( \beta \in Tr_{pass}(\mathcal{U} \cap N) \); hence, \( \beta \in Tr_{pass}(\mathcal{U}) \) and \( \beta \in Tr(N) \). As \( Tr_{pass}(\mathcal{U}) \subseteq Tr(S) \), we have that \( \beta \in Tr(S) \). It follows that \( Tr_{pass}(\mathcal{U} \cap N) = Tr_{pass}(\mathcal{U}) \cap Tr(N) \subseteq Tr_{pass}(\mathcal{U}) \cap Tr(S) = Tr_{pass}(\mathcal{U} \cap S) = Tr_{pass}(\mathcal{U}) \). Hence, \( Tr_{pass}(\mathcal{U} \cap N) \subseteq Tr_{pass}(\mathcal{U}) \). As a result, \( N \) passes each test of \( TS \), and \( TS \) is thus sound for \( S \) in \( IEIOTS(I,O,k) \) for the ioco relation.

We now prove by contradiction that \( TS \) is exhaustive for \( S \) in \( IEIOTS(I,O,k) \). Assume that \( TS \) is not exhaustive \( S \) in \( IEIOTS(I,O,k) \); thus, there exists \( N \in IEIOTS(I,O,k) \), such that \( N \text{ioco} S \) and \( N \) passes \( TS \). As \( N \) passes \( TS \), by Lemma 6, we have that \( N \) is input-state homeomorphic to \( S \); thus, by Corollary 2, \( N \) is input-state isomorphic to an initially connected submachine of \( S \) with \( k \) input states; hence, by Lemma 5, \( N \text{ioco} S \), a contradiction. We conclude then that \( TS \) is exhaustive for \( S \) in \( IEIOTS(I,O,k) \).

Therefore, \( TS \) is complete for \( S \) in \( IEIOTS(I,O,k) \) w.r.t. the ioco relation.

### 4 Concluding Remarks

In this paper, we have investigated whether it is possible to construct a finite test suite for a given IOTS specification which is complete in a predefined fault domain for the classical ioco relation even in the presence of input/output conflicts. Our conclusion is that it is in fact possible; however, under a number of assumptions about the implementations and the specifications. We have proposed a generation method which produces a finite test suite, which is complete for a given fault domain. The issue of conflicts between inputs and outputs is tackled by assuming that the implementation is “eager” to read inputs and thus such conflict is solved in favor of input, i.e., outputs are produced only if no input is presented to the implementation.

The proposed generation method is based on a classical FSM method. Thus, we rephrased the notions related to FSM generation methods, such as state cover, transition cover, state identifier, to the IOTS model. The method applies to IOTS that is minimal in the sense defined in the paper and each input state is reachable in any ioco-conforming implementation. A remarkable feature of the method is that it requires no assumption about distinguishability of output states or about their number in the specification and any implementation. Also no bound on the buffer’s length in the implementation is required to generate a complete test suite.

Our future work will focus on extending the class of IOTSs for which the approach is applicable by relaxing the mentioned constraints.

### Acknowledgment

The first author would like to thank Brazilian Funding Agency FAPESP for its partial financial support (Grant 12/02232-3). We would like to thank the anonymous reviewers for the suggestions that helped improving the paper.
References


