# Table of Contents

Preface .................................................................................................................. 1

Using Model Checking to Generate Test Cases for Android Applications ................. 7  
   Ana Rosario Espada, María del Mar Gallardo, Alberto Salmerón and Pedro Merino

ioco theory for probabilistic automata ........................................................................ 23
   Marcus Gerhold and Mariëlle Stoelinga

A Visual Formalism for Interacting Systems ................................................................. 41
   Paul C. Jorgensen

Potential Errors and Test Assessment in Software Product Line Engineering ............... 57
   Hartmut Lackner and Martin Schmidt

Adaptive Homing is in P ............................................................................................. 73
   Natalia Kushik and Nina Yevtushenko
Preface

Alexander K. Petrenko
ISPRAS
Moscow, Russia
petrenko@ispras.ru

Bernd-Holger Schlingloff
Humboldt Universität zu Berlin
Berlin, Germany
hs@informatik.hu-berlin.de

Nikolay Pakulin
ISPRAS
Moscow, Russia
npak@ispras.ru

This volume contains the proceedings of the 10th Workshop on Model-Based Testing (MBT 2015), held in London on April 18th, 2015, as a satellite workshop of the European Joint Conferences on Theory and Practice of Software (ETAPS 2015). The first Workshop on Model-Based Testing in this series took place in 2004, in Barcelona.

A tenth anniversary is a good opportunity to look back and to strike a balance, analyzing tendencies of investigations in MBT for the last 10+ years. This preface is not aimed at a thorough analysis; a flavor of the topics under consideration can be tasted by reviewing the invited papers made at the previous MBT workshops. The list of speakers and titles of their talks was as follows:

<table>
<thead>
<tr>
<th>Year</th>
<th>Speaker(s)</th>
<th>Affiliation</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>2004</td>
<td>Keith Stobie</td>
<td>Microsoft</td>
<td>Model Based Testing in Practice at Microsoft</td>
</tr>
<tr>
<td>2004</td>
<td>Rober V. Binder</td>
<td>mVerify</td>
<td>Test automation</td>
</tr>
<tr>
<td>2006</td>
<td>Harry Robinson</td>
<td>Microsoft</td>
<td>Model-based Testing for the Masses</td>
</tr>
<tr>
<td>2006</td>
<td>Alan Hartman</td>
<td>IBM Haifa Research Labs</td>
<td>Ten Years of Model Based Testing – A sober evaluation</td>
</tr>
<tr>
<td>2007</td>
<td>Rob Hierons</td>
<td>Brunel University</td>
<td>Finding a Good Order for Applying Adaptive Test Cases</td>
</tr>
<tr>
<td>2007</td>
<td>Antti Huima</td>
<td>Conformiq</td>
<td>Model-Based Testing in the Standardization of Information and Communication Technologies: the ETSI Perspective</td>
</tr>
<tr>
<td>2008</td>
<td>Wolfgang Grieskamp</td>
<td>Microsoft</td>
<td>Using Model-Based Testing for Quality Assurance of Protocol Documentation</td>
</tr>
<tr>
<td>2008</td>
<td>Marie-Claude Gaudel</td>
<td>Université de Paris-Sud, Orsay</td>
<td>Coverage-Based Random Exploration of Large Models</td>
</tr>
<tr>
<td>2009</td>
<td>Patrice Godefroid</td>
<td>Microsoft</td>
<td>Whitebox Fuzzing for Security Testing</td>
</tr>
<tr>
<td>2009</td>
<td>Darko Marinov</td>
<td>University of Illinois at Urbana-Champaign</td>
<td>Model-Based Testing Using Test Abstractions</td>
</tr>
<tr>
<td>2010</td>
<td>Jan Tretmans</td>
<td>Radboud University Nijmegen</td>
<td>Model Based System Development</td>
</tr>
<tr>
<td>2012</td>
<td>Ina Schieferdecker</td>
<td>Fraunhofer FOKUS</td>
<td>Model-Based Security Testing</td>
</tr>
<tr>
<td>2013</td>
<td>Jan Peleska</td>
<td>University of Bremen, Verified Systems International GmbH</td>
<td>Industrial-Strength Model-Based Testing – State of the Art and Current Challenges</td>
</tr>
</tbody>
</table>
We would like to give a short retrospective on MBT beginning with a talk of Alan Harman, made in 2006, because of its notable title: *Ten Years of Model Based Testing – A sober evaluation* [6]. The title of the talk claims that model-based testing has started at about 1996. In fact, MBT appeared already in the 1970s, and, apparently, even earlier. The main MBT applications in those years were in the field of testing hardware logic and telecommunication protocols. The main modeling paradigms in this field were Finite-State machine, Petri nets and other transition systems. The late 1990s is the period when the community understood that this approach can be effectively used not only for a rather narrow class of tasks close to telecommunication protocols, but also for practically all types of software, including operating systems, compilers, DBMS and others. This new approach to MBT immediately set the task to seek more adequate modeling paradigms for a new class of target systems. As the result the developers began to investigate requirement specification in the form of assertions or constraints and their special cases for program contracts.

A formal specification of requirements can be treated as a model of the behavior of a system under test. The testing task can be considered as the task of generating tests which properly cover the requirements model, with a subsequent “translation” of these “model tests” to the platform of “implementation” or “target tests”.

Apparently, this new trend of MBT applications was reflected in articles such as T. J. Ostrand and M. J. Balcer’s *The Category-Partition Method for Specifying and Generating Functional Tests* [7]. This method later became known as CPM. The essence of the CPM method is to design tests on the basis of a certain state machine, the states of which correspond to the domains of input space, i.e., partitions.

Those input space domains, in turn, correspond to the data domains where predicates making up the assertions of functional specification of the system under test operations take true or false values – that is, the input space is divided into partitions according to the terms described in specifications of functional requirements to operations under test.

Ostrand and Balcer offered a certain method of projecting tests. However, nothing was said about the tools supporting those methods. Probably, one of the first works aimed at designing a tool kit to support the method was the KVEST technology (I. Burdonov, A. Kossatchev, A.K. Petrenko, D. Galter *KVEST: Automated Generation of Test Suites from Formal Specifications* [3]) and the following UniTESK technology (I. Bourdonov, A. Kossatchev, V. Kuliamin, and A.K. Petrenko. *UniTesK Test Suite Architecture* [2]). Both technologies were based on the experience of the development testing methodology within the frames of Space Shuttle “Buran” designed in the late 1980s (A.K. Petrenko. *Test specification based on trace description* [8]).

This stage of MBT development was advanced with the emergence of test generation based on abstract state machines (W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes. *Conformance Testing with Abstract State Machines* [4]), followed by SpecExplorer (A. Blass, Y. Gurevich, L. Nachmanson, and M. Veanes. *Play to test* [1]) in 2005.

Actually, the idea of organizing an MBT workshop appeared after one of the authors of this Preface met Yury Gurevich in Novosibirsk at Ershov Conference PSI-2001. Both future organizers of the MBT workshop aimed at designing technologies for testing of system software (including operating systems). The group of A.K. Petrenko has been developing the testing approach on the basis of behavioral models in the form of program contracts. The group headed by Yury Gurevich used ASMs as the main modeling
technique. After some time it became clear that the techniques of explicit (executable) and implicit (in the form of constraints) specifications are complementary to each other. As a result, the contemporary tools UniTESK and Microsoft Spec Explorer use both model types.

At the end of 2002, the idea appeared to organize MBT workshop as a satellite event of ETAPS-2004, and devote it to testing on the basis of formal models – the idea that was gaining more and more supporters at that time. We decided not to restrict it to only a certain narrow class of models or test generation techniques. Thus, from the very first workshops there were supporters of many different variations of MBT.

The list of the invited talks, presented above, shows, on the one hand, the rather wide range of work that has been carried out in the field of MBT and, on the other hand, suggests some reflections.

In particular, it is interesting to note that in the beginning of the list, practitioners prevailed, while towards the end of the list theoreticians dominate. Why so? We could imagine a number of potential reasons:

a. Firstly, when this new theme was developing, practitioners were invited to show convincing use cases of MBT in large-scale industrial projects.

b. Secondly, it seems that in recent years practical MBT tools reached their limit in utilizing existing techniques: by now the tools have implemented most of what can be achieved at the present technological level. However, the development does not end here; we face a period of accumulating new knowledge that will make new fundamental advances in practical MBT development possible. Therefore, in recent years interest has shifted towards more theoretical research questions.

And one more observation: we can see an obvious predominance of giants such as Microsoft, IBM, Nokia among practitioners (Conformiq is mainly oriented towards Nokia’s demands). This can probably be explained by the fact that development of an industrial-strength MBT tool and MBT applications requires significant resources, which are hardly available for small and medium-sized companies.

Observing the list of invited speakers (or, more generally, the list of all the speakers at recent MBT workshops), there is no prevailing trend in the modeling paradigms or testing techniques employed. On the basis of this observation we propose that different approaches to MBT are, on the one hand, complementary, and, on the other hand, the approaches need both methodological integration and unification on the level of testing system components and unified interfaces.

This allows us to implement shared use of model analyzers and program source codes, data generators, provers, systems for collecting and run-time monitoring, etc.

Let us briefly reflect on the perspectives of MBT:

- Though no revolutionary ideas in MBT development have been proposed in recent years, there are many problems for which the development of methods and underlying theories is required.

- There are challenging technical and engineering problems in the development of tools for MBT; still there are no MBT-specific solutions, as construction of MBT tools is based on a wide range of software engineering technologies (including both traditional means and more recent ones like provers and solvers). A current trend in developing MBT tools is its integration with a variety of approaches – static and static-dynamic analysis of programs, as well as software model checking technologies.

- There are some fundamental challenges in MBT deployment:
  - The first difficulty (not the main one) is related to teaching new methods of testing for testers. This requires not only "training", but education of the tester, who also has to be an expert in
requirements analysis, their modeling and “translation” of high-level requirements presented to the level of implementation under test interfaces.

- The second problem is the issue of how to obtain “good” models. It is both technical (in what form models should be designed) and organizational (how to build a joint process of implementation design and model design). The experience shows that skilled software designers don’t want to duplicate their work and waste time on designing a model, besides the implementation itself; and unskilled programmers can do well neither in coding nor in behavior modeling.

- MBT deployment requires substantial resources, however there are no fundamental difficulties if the organization decides to introduce MBT in practice. A good example to this is the experience of Microsoft lab in China (see for instance W. Grieskamp, Xiao Qu, Xiangjun Wei, N. Kicillof, M. B. Cohen. \textit{Interaction Coverage meets Path Coverage by SMT Constraint Solving} [5]).

- MBT is actively used in the fields having the experience of systematic certification of software systems, particularly in avionics and automotive. Since this domain is becoming more and more important, a challenge is to align MBT with certification processes and regulations.

This year’s MBT workshop features Ana Cavalli from Institut National des Telecommunications, Paris, France as invited speaker. In her speech, entitled “Evolution of testing techniques: from active testing to monitoring techniques”, she presents the evolution of these testing techniques, their advantages and limitations, and illustrates the application of monitoring techniques to the security testing of real case studies.

The contributions selected by the Program Committee reflect both applications of MBT in industrial practice and further development of MBT theory and techniques.

Ana Rosario Espada, Maria Del Mar Gallardo, Alberto Salmerón and Pedro Merino present an approach to automated model construction and test generation for Android mobile applications. Marcus Gerhold and Mariëlle Stoelinga extend the well-known notion of input-output conformance to probabilistic state machines, opening the door to development of new classes of models and test construction techniques. Paul Jorgensen presents a novel variation of Petri-nets to facilitate visual modeling of interacting components in complex systems. Hartmut Lackner and Martin Schmidt discuss quality of test suites for product lines and develop an assessment approach based on mutation operators applied to software product lines. Natalia Kushik and Nina Yevtushenko present new result in the theory of FSM; they show that for some FSMs its homing sequence can be built in polynomial time.

The workshop is concluded by a remotely presented talk by Yury Gurevich on “Testing Philosophy”.

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

Alexander K. Petrenko, Holger Schlingloff, Nikolay Pakulin
March 2015

References


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 München, 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)
- Nikolay 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)
Using Model Checking to Generate Test Cases for Android Applications

Ana Rosario Espada, María del Mar Gallardo
Alberto Salmerón, Pedro Merino
Dept. Lenguajes y Ciencias de la Computación
E.T.S.I. Informática University of Málaga
[anarosario, gallardo, salmeron, pedro]@lcc.uma.es

The behavior of mobile devices is highly non-deterministic and barely predictable due to the interaction of the user with its applications. In consequence, analyzing the correctness of applications running on a smartphone involves dealing with the complexity of its environment. In this paper, we propose the use of model-based testing to describe the potential behaviors of users interacting with mobile applications. These behaviors are modeled by composing specially-designed state machines. These composed state machines can be exhaustively explored using a model checking tool to automatically generate all possible user interactions. Each generated trace model checker can be interpreted as a test case to drive a runtime analysis of actual applications. We have implemented a tool that follows the proposed methodology to analyze ANDROID devices using the model checker SPIN as the exhaustive generator of test cases.

1 Introduction

At present, smartphone technology is ubiquitous and changes constantly. Users use their mobiles not only as phones, but as compact computers, able to concurrently provide services which are rapidly created, updated, renewed and distributed. In this scenario of continuous evolution, different operating systems have been developed such as SYMBIAM, IOS, WINDOWS PHONE and ANDROID, which allow phones to support more and more complex applications. These platforms define new models of execution, quite different from those used by non-mobile devices. For instance, one of the most defining characteristics of these systems is their open and event-driven nature. Mobile devices execute a continuous cycle that consists of first, waiting for the user input and second, producing a response according to that input. In addition, the internal structure of mobile systems is constructed from a complex combination of applications, which enable users to easily navigate through them. Thus, although, at a lower level, the execution of applications on a mobile device involves the concurrent execution of several processes (for instance, in ANDROID, applications are JAVA processes executing on the underlying LINUX operating system), the way these applications interact with each other and with the environment does not correspond with the standard interleaving based concurrency model.

It is clear that the execution of applications on these new operating systems, such as ANDROID [1], may lead to the appearance of undesirable bugs which may cause the phone to malfunction. For example, mobile devices may display the typical errors of concurrent systems such as violations of safety and liveness properties. However, there are other bugs inherent to the particular concurrency model supported by the new platforms which are not directly analyzable using current verification technologies. For

---

*Work partially supported by grants P11-TIC-07659 (Regional Government of Andalusia), TIN2012-35669 (Spanish Ministry of Economy and Competitiveness), UMA-Agilent Technologies 808/47.3868- 4Green and the AUIP as sponsor of the Scholarship Program Academic Mobility.

---

Espada, Gallardo, Salmerón & Merino
example, applications could incorrectly implement the life cycles of their activities or services (in the case of ANDROID), or may misbehave upon the arrival of unexpected external events. In addition, conversion errors, unhandled exceptions, errors of incompatibility API and I/O interaction errors as described in [16] may also appear.

Different techniques for analyzing the execution of mobile platforms have been proposed. Verification approaches such as model checking [9] can be applied to the software for mobile devices [22, 21, 19]. Model checking is based on an exhaustive generation of all the interleavings for the threads/processes. A major problem to apply this technique to the real code, like mobile applications, is the need to construct a model of the underlying operating system or libraries [10, 7, 11]. The open nature of these platforms, which are continuously interacting with an unspecified environment, makes other analysis techniques such as testing, monitoring, and runtime verification more suitable to check bugs without too much extra effort to model the operating system or the libraries. There have been several recent proposals [12, 24, 18] for testing in this framework with commercial tools [4, 2]. In these approaches, test cases are randomly generated with tools such as MONKEY and MONKEYRUNNER [1].

Testing and runtime verification maybe also combined, as described in [6], to construct verification tools for mobile applications [20, 26]. On the one hand, the careful selection of test cases guides the execution of the device, while, on the other, the runtime verification module implements observers devoted to analyzing the traces produced by the device. The runtime verification module was already addressed by some authors of this paper in [14]. Here we focus on describing how the generation of test cases may be carried out following the model-based testing approach [25] supported by model checking tools.

Our proposal is based on the idea that although the interaction between the user and the mobile device is completely undetermined, each application is associated with a set of intended user behaviors which define the common ways of using the application. For each application, or more precisely, for each application view, we use state machines to construct a non deterministic model representing the expected use of the view/application. This state machine is built semiautomatically, with information provided by the expert (the app designer or tester) and by ANDROID supporting tools like UIAUTOMATORVIEWER. Then, all these view models may be conveniently composed to construct a non deterministic model of the user interaction with a subset of mobile applications of interest. Due to the way of building the state machines, each execution trace of the composed state machine corresponds to a possible realistic use of the mobile. Thus, the generation of test cases is reduced to the generation of all possible behaviors of the composed machine, which may be carried out by a model checking tool. Although the methodology proposed does not depend on the underlying mobile operating system, the tool has been built on the assumption that the operating system is ANDROID.

The paper provides two main contributions. The first one is the formal definition of a special type of state machine that models the expected user interaction with the mobile application. The approach to modeling is completely modular in the sense that adding (or removing) new view state machines to incorporate (eliminate) user behaviors does not affect the rest of state machines that have already been defined. The second one is a method to employ the explicit model checker SPIN [15] that takes the composed state machine as input and produces a significant set of test cases that generate traces for runtime verification tools. We have constructed a tool chain which implements both modeling and test generation phases to shows the feasibility of the approach in practice.

The rest of the paper is organized as follows. Section 2 describes our approach to using model checking for test case generation. Section 3 introduces the testing platform that we are developing. Section 4 provides a formal description of the behaviour of composed state machines. Section 5 uses well known ANDROID applications to describe how our approach for test case generation is implemented. Section 6 gives a comparison with related work. Last section summarizes conclusions and future work.
mtype = { state_init, state_1, state_2 };
typedef Device { byte transitions[MAX_TR]; short index; bool finish; }
Device devices[DEVICES];
mtype state[DEVICES];
active proctype traceCloser() provided ( devices[DEVA].finish && devices[DEVB].finish ) {
    end_tc: outputTransitions()
}
active proctype device_A() {
    state[DEVA] = state_init;
    do
        :: state[DEVA] == state_init -> transition(DEVA, BUTTON_1); state[DEVA] = state_1
        :: state[DEVA] == state_1 -> transition(DEVA, SWIPE); state[DEVA] = state_1
        :: state[DEVA] == state_1 -> transition(DEVA, BUTTON_2); state[DEVA] = state_2
        :: state[DEVA] == state_2 -> transition(DEVA, MESSAGE); break
        :: state[DEVA] == state_2 -> transition(DEVA, BACK); break
    od;
    devices[DEVA].finish = true;
}
active proctype device_B() {
    state[DEVB] = state_init;
    ...
    devices[DEVB].finish = true;
}

Listing 1: Sample PROMELA specification for test generation

2 Model checking for test case generation

SPIN [15] is a model checker that can be used to verify the correctness of concurrent software systems modeled using the specification language PROMELA. The focus of the tool is on the design and validation of computer protocols, although it has been applied to other areas. SPIN can check the occurrence of a property over all possible executions of a system specification, and provide counterexamples when violations are found.

We use the SPIN model checker in our approach for automatically generating test cases from application models in the following way. First, each device will be represented by a single PROMELA process, which contains a state machine that models the applications contained on that device. The state machines themselves are written as loops, where each branch corresponds to a transition triggered by an event. The current state of each state machine (stored as a global PROMELA variable) determines which branches are active and may be taken. The right hand side of each branch records the transition and updates the current state. This PROMELA specification is explored exhaustively by SPIN in order to generate all possible test cases described by the application model, taking all possible alternatives when there is more than one active branch at the same time.

Listing 1 shows an example of a PROMELA specification that follows the approach outlined above. This example contains two devices and with their corresponding state machine (device_A() in line 8 and device_B() in line 19), with two states plus the initial state. The transition function is used to record the user or system transition associated with each branch. In order to complete a test case, all devices must have finished their respective state machines (lines 17 and 22), usually when the do loop is exited (lines 14 and 15). This enables the traceCloser process to be executed due to its schedulability restrictions (line 5), which prints the transitions of the generated test case.

In addition to the current state (line 4), this PROMELA specification also keeps a list of the transitions taken on the test currently being generated (line 2). The purpose of this data structure is twofold. On the one hand, outputTransitions will print the trace stored here. On the other hand, the history of the
current trace is kept inside the SPIN’s global state, which is taken into consideration when deciding if a state has already been visited. Thus, the same transition may be taken more than once if possible (e.g. line 12), since the history of the states will be different. However, this requires the maximum depth of exploration to be bounded by the MAX_TR constant (line 2).

3 Architecture of the platform

Figure 1 shows the general structure of tools that combine testing and runtime verification techniques to analyze the behavior of applications running on mobile devices. The bottom side uses observers/monitors to analyze the resulting execution traces and verify whether they comply with the expected properties as implemented in the tool DRAGONFLY [14, 13]. The top side of the figure shows the generation of test cases considered in this paper. The Tester is the expert responsible for modeling the behavior of the applications to be analyzed using a state chart diagram. These models may be constructed as part of the design phase of the applications, and are characterized by their compositional nature: functionality can be added to an existing view without essentially altering the existing behavior.

Figure 2 shows the complete process of our actual proposal for test generation and execution, which is divided into three main modules:

- **Modeling.** UIAUTOMATOR VIEWER tool from ANDROID tools extracts the controls definition in each view of the ANDROID application under analysis. Then, the controls definition and the state chart diagrams are associated into a Model.xml file with a given structure.

- **Test Case Generation.** Creates a test case generator per XML file model into a PROMELA file. The SPIN model checker [15] performs an exhaustive search of all valid paths in the model using the method explained in Section 2, which correspond to test cases, and generates an XML file for each one with the appropriate sequence of user input events.

- **Test Case Execution.** Generates each test class provided using the valid paths described into XML file by the test case generating module. Then, they can be executed by the ANDROID framework, and sends them to the devices to be executed using the UIAUTOMATOR tool which is an extension of JUnit tool using to write user interfaces test cases for ANDROID.

The following sections provide details about the internal behavior of the Modeling and Test Case Generator modules, which are the aim of this work.
4 Formal Description of models

In the following description, we define the behaviour of mobile applications through the composition of state machines at different abstraction levels. The lowest level is composed of view state machines. A view corresponds to a mobile screen, with its buttons, text fields, etc., through which users may interact with the device. When the view is active, users may fire events through its interface. A view state machine models the possible behaviors of the user when he/she is making use of the view. These behaviors coincide with the sequence of events fired by the user. Sometimes one of these events makes a different view becomes active. We have modeled this control transfer between views through the composition relation of view state machines from which device state machines are constructed. Device state machines use the connection states to switch from the current active view to a different view. In this formalization, the specific applications to which each view belongs have not been taken into account, that is, we only model the transfer from one view to another, irrespective of whether both views belong to the same application. In the sequel, we use symbols $\rightarrow / \rightarrow_i$ to denote the transition relation of the view state machines $M_i$. In addition, symbol $\rightarrow_c$ defines the binary relation which allows us to connect view state machines. Finally, $\rightarrow_d$ represents the transition relation of the device state machine which is constructed from relations $\rightarrow / \rightarrow_i$ and $\rightarrow_c$.

Since ANDROID applications are event driven, we may consider that each test case corresponds to the sequence of events fired which drive the mobile behaviour. In the formal description, events are the labels of transitions ($\rightarrow / \rightarrow_i$, $\rightarrow_c$, $\rightarrow_d$) and have the natural meaning. For instance, $s \xrightarrow{e} s'$ means that event $e$ must be fired to be able to transit from $s$ to $s'$.

4.1 View state machines

Definition 1 A view state machine is a tuple $M = (\Sigma, I, \rightarrow, E, C, F)$, where $\Sigma$ is a finite set of states, $I \subseteq \Sigma$ are the initial states, $C \subseteq \Sigma$ are the so-called connection states, $F \subseteq \Sigma$ is the set of final states, $E$ is the set of user events, and $\rightarrow \subseteq \Sigma \times E \times \Sigma$ is the labelled transition relation. Sets $I, C$ and $F$ are mutually disjoint.

Final states are states from which it is not possible to evolve. Connection states are states from which it is possible to transit a different state machine. These states are essential to model the switch between
typical views of smart phone devices. Usually, when a new view is called, the execution of the system is supposed to return to the view caller. To take this behavior into account, we assume that each connection state \( s \in C \) has a related state \( return(s) \in \Sigma \) which represents the state to be returned when the new view invoked from \( s \) has finished its execution.

We partition set of events \( E \) into two disjunct sets: the set of user events, denoted as \( E^+ \), which contains events such as pressing a button, etc., and the set of system events, denoted as \( E^- \), which includes, for instance, events corresponding to system responses to user requests. In the following, we use \( e^+ \), \( e^- \) to represent user events and system events, and we use \( e \) to refer to events which may be of any of both types.

View state machines are deterministic in the sense that if \( s \xrightarrow{e^+} s_1 \), and \( s \xrightarrow{e^-} s_2 \) and \( e = e' \), then \( s_1 = s_2 \). That is, the machine defines, at most, a transition for each pair state/input event.

We now define the notion of flow (an execution in a view state machine), and the test cases generated from flows.

**Definition 2** Given a view state machine \( M = \langle \Sigma, I, \xrightarrow{\rightarrow}, E, C, F \rangle \), we define the set \( Flow(M) = \{s_0 \xrightarrow{e_1} s_1 \xrightarrow{e_2} \cdots s_n \xrightarrow{e_n} s_0 | s_0 \in I, s_n \in F \cup C \} \) of all sequences of states, allowed by \( M \), starting at an initial state of \( M \), and ending at a final or connection state of \( M \). The length of a flow is the number of its states. Given a flow of length \( n \), \( \phi = s_0 \xrightarrow{e_1} \cdots \xrightarrow{e_n} s_n \in Flow(M) \), the sequence of events determined by \( \phi \) (the test case) is \( test(\phi) = e_1 \cdots e_n \). We define the set of test cases allowed by \( M \) as \( TC(M) = \{test(\phi) | \phi \in Flow(M)\} \).

According with Definition 2, test cases are finite sequences of user and system events. For instance, sequence \( e_1^+ \cdot e_2^- \cdot e_3^- \cdot e_4^+ \) represents a test case where the user first fires events \( e_1^+ \) and \( e_2^- \), then the system fires \( e_3^- \), and finally user fires \( e_4^+ \). Thus, user and system events are similarly dealt with during the generation of test cases. The difference between them is of importance when test cases are transformed into code to be executed on the mobile as described in Section 5. User events will be transformed into non-blocking calls to methods that simulate the real occurrence of the event, while system events will correspond to calls to blocking methods which wait for the arrival of the system event.

### 4.2 Composition of view state machines

In this section, we describe how view state machines are composed to construct flows that navigate through different views representing realistic ways of using a mobile.

We first define the transition between different view state machines. This transition is realized through the binary relation \( \mathcal{R} \) defined between the connection and initial states. The idea is as follows. Assume that the flow in execution belongs to a view state machine \( M_i \), and that a connection state \( cs \) of \( M_i \) has been reached. If relation \( \mathcal{R} \) defines a transition from \( cs \) to some initial state of other machine \( M_j \), the flow could jump from \( M_i \) to \( M_j \), and proceed following the transition relation of \( M_j \). This jump implies the change in the activity visible in the device from \( M_i \) to \( M_j \). In the sequel, we call active the view state machine which is visible in the device, and create to the rest of view state machines which have been created but are not currently visible in the device.

Given a finite family of state machines \( M_i = \langle \Sigma_i, I_i, \xrightarrow{\rightarrow}, E_i, C_i, F_i \rangle \) we define \( \Sigma = \cup_{i=1}^n \Sigma_i, I = \cup_{i=1}^n I_i, E = \cup_{i=1}^n E_i, C = \cup_{i=1}^n C_i, \) and \( F = \cup_{i=1}^n F_i \). In addition, we denote with \( \mathcal{E} \subseteq E \) the set of call events that provoke the switch between active view state machines.

**Definition 3** Let us assume a finite family of state machines, \( M_i = \langle \Sigma_i, I_i, \xrightarrow{\rightarrow}, E_i, C_i, F_i \rangle \). The connection of view state machines \( M_1, \cdots, M_n \) is given by a binary relation \( \mathcal{R}(M_1, \cdots, M_n) \subseteq C \times \mathcal{E} \times I \), that connects connection states with initial states. In the following, we denote 3-uples \( (s_i, e, s_j) \) of \( \mathcal{R}(M_1, \cdots, M_n) \) as \( s_i \xrightarrow{e} s_j \). Observe that source and target machines \( i \) and \( j \) may coincide.
When a new view is created, the call event may specify some parameters that determine how it must be started or finished. For instance, if the view has already been created, the caller may choose whether to reuse the previously created view or, to the contrary, create a new one. Additionally, when the new created view has finished the execution, the caller view may automatically become active or not. Boolean functions \(\text{reuse}, \text{auto\_return} : \epsilon \rightarrow \{\text{false}, \text{true}\}\) establish these parameters for the call events. Although there are other parameters that can be defined in the call events, these two are sufficient to describe the mobile behaviour.

We now define the device state machine which composes the behavior displayed by the view state machines using the connection relation.

**Definition 4** Let us assume a finite family of state machines, \(M_i = (\Sigma_i, I_i, \rightarrow_i, E_i, C_i, F_i)\), and a connection relation of \(M_1, \cdots, M_n, \mathcal{R}(M_1, \cdots, M_n)\), as defined above. The device state machine

\[
\mathcal{D} = M_1 \| \cdots \| M_n \| \mathcal{R}(M_1, \cdots, M_n)
\]

is defined as the state machine \((\Sigma \times \Sigma^* \times \epsilon^*, I, \rightarrow_d, E, F)\) where

1. \(\Sigma^*\) is the set of finite sequences of states of \(\Sigma\), and \(\epsilon^*\) is the set of finite sequences of call events.

2. Transition relation \(\rightarrow_d\) is defined by the rules below.

We call configurations the states of device state machines. A configuration is a 3-uple \((s, h, eh)\) where \(s\) is the current state of the active view state machine, i.e., the view visible in the mobile. Sequence \(h\) is the stack of states \(s_1 \cdot s_2 \cdots s_n\) which constitute the history of the view machines which have been created (and have not been yet destroyed) in the device but which are not currently visible. Each state \(s_i\) of \(s_1 \cdot s_2 \cdots s_n\) is a connection state of a view state machine which was active, but a transition from \(s_i\) to another view machine took place, and the view state machine of \(s_i\) became inactive. Finally, \(eh = e_1 \cdots e_n\) is the history of events that have provoked a view switch in the current execution. Thus, \(e_i \in \epsilon\) is the event which fired the transition from state \(s_i\) to an initial state of another view state machine. In the following, \(\epsilon\) represents the empty (event) history.

The evolution of configurations is given by the transition relation \(\rightarrow_d\) defined by the rules in Figure 3. Relation \(\rightarrow_d\) is constructed from the transition relations of view state machines \(\rightarrow_i\), and the binary connection relation \(\rightarrow_c\). In these rules, given a history of states \(s_1 \cdots s_n\) and the index \(j\) of a view state machine \(M_j\), function \(\text{top} : \Sigma^* \times \mathcal{N} \rightarrow \Sigma \cup \{\bot\}\) returns the last state of the view state machine \(M_j\) in the sequence \(s_1 \cdots s_n\). That is, \(\text{top}(s_1 \cdots s_n, j)\) returns \(s_k\), if \(1 \leq k \leq n\) is the biggest index such that \(s_k \in \Sigma_j\), or \(\bot\), if such a state does not exist.

Rule \(\text{R1}\) states that a transition inside a view state machine \(M_i\) corresponds to a transition in the device state machine. Rules \(\text{R2}, \text{R3}\) model a transition from machine \(M_i\) to machine \(M_j\) when both the new state \(s'\) and the event \(e\) are added to the state and event histories of the current system configuration. Rule \(\text{R2}\) is applied when event \(e\) does not involve reusing a previously created view (\(\text{reuse}(e)\) is false), while \(\text{R3}\) applies when a view of \(M_j\) should have been reused (\(\text{reuse}(e)\) is true), but the current state history does not contain one \(\text{top}(s_1 \cdots s_n, j) = \bot\). Rule \(\text{R4}\) defines a transition from machine \(M_i\) to \(M_j\) by reusing a previously created flow of \(M_j\) (\(\text{reuse}(e)\) is true) which is stored in the configuration history \(\text{top}(s_1 \cdots s_n, j) = s_k\). Finally, \(\text{R5}\) defines the case when the flow of the current active view has finished, and the execution must continue with the view stored at the top of the state history. Otherwise, that is, if \(\text{auto\_return}(e)\) returns false, the current configuration \((s, h, eh)\) cannot evolve.
Using Model Checking to Generate Test Cases for Android Applications

R1. \( s \xrightarrow{e} s' \)
\( \langle s, h, eh \rangle \xrightarrow{e} \langle s', h, eh \rangle \)

R2. \( s \in C_i, s \xrightarrow{e} s', \neg \text{reuse}(e) \)
\( \langle s, h, eh \rangle \xrightarrow{e} \langle s', h \cdot \text{return}(s), eh \cdot e \rangle \)

R3. \( s \in C_i, s' \in I_j, s \xrightarrow{e} s', \text{reuse}(e), \text{top}(s_1 \cdots s_n, j) = 1 \)
\( \langle s, h, eh \rangle \xrightarrow{e} \langle s', h \cdot \text{return}(s), eh \cdot e \rangle \)

R4. \( s \in C_i, s' \in I_j, s \xrightarrow{e} s', \text{reuse}(e), \text{top}(s_1 \cdots s_n, j) = s_k \)
\( \langle s, s_1 \cdots s_n, e_1 \cdots e_n \rangle \xrightarrow{e} \langle s_k, s_1 \cdots s_{k-1}, e_1 \cdots e_{k-1} \rangle \)

R5. \( s \in F_i, \text{auto}\_\text{return}(e) \)
\( \langle s, h \cdot s', eh \cdot e \rangle \xrightarrow{e} \langle s, h, eh \rangle \)

R6. \( c_0 \xrightarrow{e} c_1 \)
\( \langle c_0, c'_0, dh \rangle \xrightarrow{e} \langle c_1, c'_0, dh + \{e^+\} \rangle \)

R7. \( c'_1 \xrightarrow{e} c'_1 \), \( e^+ \in \text{dh} \)
\( \langle c_0, c'_0, dh \rangle \xrightarrow{e} \langle c_1, c'_1, dh - \{e^+\} \rangle \)

Figure 3: Transition relation rules

Definition 5 Given a device state machine
\[ D = M_1 || \cdots || M_n || \mathcal{R}(M_1, \cdots, M_n) \]
\[ = (\Sigma \times \Sigma^* \times \mathcal{E}^*, I, \rightarrow_d, E \cup \mathcal{E}, F) \]

1. the trace-based semantics determined by \( D (\mathcal{O}(D)) \) is given by the set of finite/infinite sequences of configurations (flows) produced by the transition relation \( \rightarrow_d \) from an initial state, that is,
\( \mathcal{O}(D) = \{ \langle s_0, e, e \rangle \xrightarrow{e_0} \langle s_1, h_1, eh_1 \rangle \cdots | s_0 \in I \} \).

2. Given a flow \( \phi = c_0 \xrightarrow{e_1} c_1 \xrightarrow{e_2} c_2 \cdots \in \mathcal{O}(D) \), the test case determined by \( \phi \) is the sequence of events test(\( \phi \)) = \( e_1 \cdot e_2 \cdots \).

3. The set of test cases determined by a set of flows \( \mathcal{T} \) is TC(\( \mathcal{T} \)) = \{ test(t) | t \in \mathcal{T} \} \).

Thus, a flow \( \phi \in \mathcal{O}(D) \) consists of a sequence of view state machine flows (Definition 2) connected throw connection states. Flow \( \phi \) may finish at a final state of some view state machine, or may be infinite. The length \( |\phi| \) of a flow is the number of its states (configurations), if it is finite, or \( \infty \), otherwise. Given a flow \( \phi = c_0 \xrightarrow{e_1} c_1 \xrightarrow{e_2} c_2 \cdots \in \mathcal{O}(D) \), we define the truncated flow of \( n \), \( \phi^n \), as \( \phi \) iff \( |\phi| \leq n \) or \( \phi^n = c_0 \xrightarrow{e_1} c_1 \xrightarrow{e_2} c_2 \cdots \xrightarrow{e_{n-1}} c_{n-1} \), otherwise. Considering this, we define the set of traces \( \mathcal{O}^n(D) \) as the set all traces of \( \mathcal{O}(D) \) truncated up to length \( n \), that is, \( \mathcal{O}^n(D) = \{ \phi^n | \phi \in \mathcal{O}(D) \} \).

Observe that the state space of device state machines is not finite because configurations include the state and event histories which may have arbitrary lengths. In addition, the state space generated when an explicit model checker is constructing all the flows allowed by a device state machine is non-finite not only due to the state and event histories, but also because the matching algorithm, carried out during the state space search, must take into account both the current state of the flow and the history of the previous states of the flow. This allows that, for instance, flows \( \phi_1 = \langle s_0, e, e \rangle \xrightarrow{e_1} \langle s_1, e, e \rangle \xrightarrow{e_2} \langle s_2, e, e \rangle \xrightarrow{e_3} \langle s_3, e, e \rangle \) and \( \phi_2 = \langle s_0, e, e \rangle \xrightarrow{e_4} \langle s_4, e, e \rangle \xrightarrow{e_5} \langle s_5, e, e \rangle \) can be both generated by the model checker although when constructing \( \phi_2 \) state \( s_1 \) has been already visited as explained in Section 2.
In consequence, the models of device state machines are not, in general, state finite which means that, the model checking process does not, in general, terminate. In the current implementation, we have solved this problem by bounding the depth of the execution flows analyzed generating $O(n)(D)$ for some fixed $n$.

4.2.1 Composing several devices

The extension of the state machine model to several devices is carried out by composing the device state machines by interleaving. Thus, if $c_0 \xrightarrow{e_1} c_1$ and $c_0' \xrightarrow{e_1'} c_1'$ are a transitions in devices $D$ and $D'$, respectively, then they allow the two following transitions, $(c_0, c_0') \xrightarrow{e_1} (c_1, c_1')$ and $(c_0, c_0') \xrightarrow{e_1'} (c_0, c_1')$ in the interleaved composition of $D$ and $D'$. The communication between both devices is modeled by a user event in the sender device (the device that starts the communication), and a system event in the receiver device (the device that expects the message).

This is described in the last two rules of Figure 3. Rule R6 handles the transition from the sender, and R7 handles the transition in the receiver. Note that $dh$ denotes the set of system events produced but not yet consumed. Thus, for instance, using the previous example, if $e_1 = e_1^+$ is an event that implies a communication from $D$ to $D'$, and $e_1' = e_1^-$ is the corresponding event to be read by $D'$ from $D$, we would generate the test cases $e_1^+ \cdot e_1^-$ and $e_1^- \cdot e_1^+$. Note that in the second test case, the method that implements the transition for the receiver event will suspend the execution of $D'$ until event $e_1^+$ is fired by $D$.

In addition, when dealing with more that one device, we make use of model checking optimization techniques such as partial order reduction [15] to avoid the generation of different test cases that correspond to a single feasible interaction between the devices.

5 Case Study

In this section, we describe how the behavior of mobile applications is modeled and how tests cases are automatically generated from these models.

5.1 Modeling

We first need to construct an abstract model of the system to be analyzed, using statecharts and following the notions of view and device state machines given in Section 4. This model should include the relevant user interactions for the tests we want to perform. For instance, a test case which is affected by whether the GPS is on or off may include user interactions to change its status, while other tests may not need those interactions. A test case generator will be created from this model using automatic transformations. This modeling step can be performed separately from the design of the application, or in combination as is custom in other model-driven tools such as IBM Rational Rhapsody [3]. In addition, the controls of each screen have to be extracted and modeled, so that transitions on the state machines can be tied to actions performed on these controls.

We use a scenario with two applications, Facebook and YouTube. This scenario is composed of three views (HomeView, CommentView and MovieView), which describe the behavior of a user placing a link to a YouTube video in a Facebook comment, and watching this or other videos in YouTube. The state machines can be modeled using UML as shown in Figure 4. These state machines include additional information required to correlate them with the applications and their views. An XML definition of the model can then be automatically generated from these state machines. Listing 2 shows part of this XML.
definition \(^1\). In particular, it contains the state machine associated with the *Home* view of the Facebook app. Each state machine may define several states and transitions. In addition to simple transitions between states of the same state machine, it is also possible to define transitions that call another state machine and, upon its termination, continue in the caller machine. The *type* and *through* attributes identify the type of transition and the state machine to call (if any). Listing 2 provides examples of both simple (line 10) and complex (lines 9 and 11) transitions. Each transition has an unique ID within its view that is used to identify transitions, and also declares the user or system event that triggers it.

The events that fire the transitions in Figure 4 are the user actions performed on controls placed in visible views. We organize controls into *group of controls* according to the actions associated with each. Figure 5 shows some of the control groups that have been identified in the *CommentView* View. For instance, the *Comment* group could represent any of the text fields to write a comment, and the *clickYouTubeLink* identifies links to YouTube videos.

These control groups are declared in a *control definition* file with the help of the UIAUTOMA-

---

\(^1\)More complete versions of this and other parts of the model are available online at [http://morse.uma.es](http://morse.uma.es).
<Application name="Facebook" package="com.facebook.android">
  <Views>
    <View name="HomeView" controlsFile="Home.xml">
      <StateMachine name="HomeUpdate">
        <States>
          <State name="S0"/>
          <State name="S1"/>
        </States>
        <Transitions>
          <Transition ID="1" event="Swipe" prev="S0" next="S0" type="Simple"/>
          <Transition ID="2" event="Comment" prev="S0" next="S0" through="CommentView" type="View"/>
          <Transition ID="3" event="Swipe" prev="S0" next="S1" type="Simple"/>
          <Transition ID="4" event="ClickYouTubeLink" prev="S0" next="S0" through="ViewingMovieStateMachine" type="StateMachine"/>
          <Transition ID="5" event="Swipe" prev="S1" next="S1" type="Simple"/>
          <Transition ID="6" event="Comment" prev="S1" next="S0" through="CommentView" type="View"/>
          <Transition ID="7" event="Swipe" prev="S1" next="S0" type="Simple"/>
        </Transitions>
      </StateMachine>
    </View>
  </Views>
</Application>

Listing 2: State machine configuration

<node index="0" text="" testGroup="" ....
<node index="0" ....
<node testGroup="clicLike" IsFixedValue="" PatternOrValue="" index="0" text="Like" resource-id="id/feed_feedback_like_container" clickable="true" long clickable="false" password="false" ... />

Listing 3: Control group definition

TORVIEWER tool [1]. This tool analyzes each view without requiring its source code, and generates an UIX (XML) file containing the hierarchy of controls in the view. Listing 3 shows part of the generated file for the Facebook application. The attributes associated with each control in the UIX file include the kind of actions that the control supports, such as clickable or scrollable. The UIX file is then customized to bring together the controls which belong to the same group by setting the controlGroup attribute. Some controls accept parameters which may also be included as attributes in this file. For instance, the values introduced in text fields may be fixed or generated automatically according to some pattern.

5.2 Test case generation

We are now ready to generate the corresponding test cases in an exhaustive manner. The XML file is automatically transformed into a PROMELA specification that follows the same principles described in Section 2, but with a few additions to accommodate the structure of ANDROID applications. Each device is still represented by a single process, but their state machines are defined in separate inlines, one per each app, view and state machine, which can then be composed. In addition, there may be device-specific app and view inlines, since views and state machines can be assigned to a particular device. In the simplest form of composition, device processes call their app inlines, app inlines call their view inlines, and view inlines call their state machine inlines. On the other hand, a state machine may call another view or state machine. When this happens, the state of the previous state machine should be stored such that when the new one is finished, the state is correctly restored. To support this we introduce a backstack data structure, where the state of the current state machine is always at the top of the stack.

Listing 4 shows a simplified extract of the PROMELA specification generated for the Facebook and
typedef Backstack { mtype states[MAX_BK]; short index; }
Backstack backstacks[DEVICES];
#define currentBackstack devices[device].backstack
#define currentState currentBackstack.states[currentBackstack.index]
active proctype device_219dcac41() {
  if (true -> app_219dcac41_Facebook(D_219dcac41);
   true -> app_219dcac41_YouTube(D_219dcac41);
  fi;
  devices[D_219dcac41].finished = true
}
inline statemachine_Facebook_HomeView_HomeUpdate (device) {
  currentState = State_Facebook_HomeView_HomeUpdate_init;
  pushToBackstack (device, State_Facebook_HomeView_HomeUpdate_init);
  do 
    : currentState == State_Facebook_HomeView_HomeUpdate_S0 ->
      transition (device, VIEW_HomeView, 2);
      view_Facebook_CommentView (device);
    : currentState == State_Facebook_HomeView_HomeUpdate_S0 ->
      transition (device, VIEW_HomeView, 4);
      statemachine_YouTube_MovieView_ViewingMovieStateMachine (device);
  od;
  popFromBackstack (device)
}

Listing 4: PROMELA specification for Facebook and YouTube

YouTube example. The backstack data structure is shown on line 2. A new element is pushed to or popped from the backstack at the beginning or end of a state machine, respectively (lines 14 and 26), while currentState always point to the top of this stack for each device.

Each sequence of transitions generated by SPIN is translated into a UAutomatorTestCase subclass, where each transition is implemented by a method. This class simulates the actions performed by the user, such as pressing buttons or swiping. The code shown in Listing 5 shows part of a test case obtained from the model of Figure 4, in particular a user adding a link to a YouTube video in a Facebook comment, and later watching that video on the YouTube application. The file is compiled into a .dex (ANDROID application binary) file, and then deployed into a ANDROID device and executed using the adb tool.

Table 1 provides some quantitative results of the number of test cases generated and the computational effort required, for several scenarios, averaged over three runs. Device 219dcac4 was assigned only the Facebook application, while device 219dcac41 was assigned both, although in both cases other modeled applications may be reached from the assigned ones. The fourth column declares the maximum depth allowed for the test case transitions generated for a device. The fifth column represents the total time spent to generate the test cases from the XML models. The last three columns are stats taken from the SPIN execution, namely the number of SPIN states generated, the size of each state, and the total memory spent, respectively. These results show how adding the YouTube application, which is fairly isolated, has little impact in the results (rows 3 and 4 of data).

6 Comparison with related work

There are other proposals to apply model-based testing to ANDROID applications. Some of them consider that the testing process starts without a precise model of the expected behavior of the applications and
Listing 5: Generated UiAutomatorTestCase

```java
public class TestDevice1 extends UiAutomatorTestCase {
    // Transition 2: previous S0 next S0 on view HomeView
    public void TestFacebookComment2() throws UiObjectNotFoundException {
        UiObject control = new UiObject(new UiSelector().
            className("android.widget.TextView").index(1).textContains("Comment"));
        control.click();
    }
    // Transition 4: previous S0 next S0 on view HomeView
    public void TestFacebookClicYouTubeLink27() throws UiObjectNotFoundException {
        UiObject control = new UiObject(new UiSelector().
            className("android.view.View").index(3));
        control.click();
    }
    // Transition 1: previous next Y0 on view MovieView
    public void TestYouTubeplaypause28() throws UiObjectNotFoundException {
        UiObject control = new UiObject(new UiSelector().
            className("android.view.View").index(4));
        control.click();
    }
}
```

Table 1: Test case generation results

<table>
<thead>
<tr>
<th>Devices</th>
<th>Configuration</th>
<th># Test Cases</th>
<th>Time (s)</th>
<th># States</th>
<th>State Size (B)</th>
<th>Memory (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>219dcac4</td>
<td>Backstack 4</td>
<td>5641</td>
<td>1.0</td>
<td>307234</td>
<td>84</td>
<td>156.8</td>
</tr>
<tr>
<td></td>
<td>Transitions 20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>219dcac41</td>
<td></td>
<td>111317</td>
<td>9.0</td>
<td>6063398</td>
<td>92</td>
<td>728.6</td>
</tr>
<tr>
<td>✓</td>
<td></td>
<td>5660</td>
<td>1.0</td>
<td>307493</td>
<td>84</td>
<td>156.8</td>
</tr>
<tr>
<td>✓</td>
<td></td>
<td>111342</td>
<td>9.0</td>
<td>6063735</td>
<td>92</td>
<td>728.6</td>
</tr>
<tr>
<td>✓</td>
<td></td>
<td>1872</td>
<td>7.0</td>
<td>4039337</td>
<td>100</td>
<td>560.3</td>
</tr>
<tr>
<td>✓</td>
<td></td>
<td>12180</td>
<td>52.3</td>
<td>28972472</td>
<td>108</td>
<td>3445.2</td>
</tr>
</tbody>
</table>

they focus on techniques to obtain such model. MobiGUITAR framework [5] automatically construct a state machine of one application by executing events in the running application and recording a tree with fireable events for each new state. The authors use a "breadth-first" traversal of the apps GUI for open source applications. As far as they are not considering any knowledge on the way of using the application but they are making an exhaustive execution, they need some criteria to assume whether some states are equivalent to prevent state explosion. The Swift-Hand technique proposed in [8] employs machine learning to construct an approximated model of the application during the testing process. Their aim is to cover as much behavior as possible, making the execution to enter in unexplored parts of the state space. In our method, we separate test generation from testing and the states in our high-level state machines are limited and differentiated by design. So our models are more compact, and for instance, compared with MobiGUITAR we do not need extra work to remove unrealistic test cases. In addition, our approach allows to generate test cases for several applications that interact using ANDROID intents, while the complexity of the runtime based modeling process for MobiGUITAR and Swift-Hand makes them more suitable for single applications.

Like in our proposal, other works also consider the existence of a formal specification of the applications to start the test generation. In [17] the authors describe how to follow a property-driven method build the models in Alloy, a formal language based on high order logic. In their proposal the role of the model checker in our approach is done they the Alloy analyzer, that generates positive (expected) and negative (undesired) test cases. Like in our approach, they use XML based transformations to translate
the test cases to some executable form in order to activate the applications under test. Apart from the inside technologies (model checking vs constraint solver), the main difference in both proposals is the way to obtain the refined executable model. The Alloy specification in [17] is constructed manually, while the PROMELA specification in our work is done automatically from the high level design of the view state machines. We still need to work in the same case study to get a quantitative comparison on the human and computational effort required in both approaches.

There are other model-based testing tools for Android which are not focused on models that consider the user inputs. For instance, the tool APSET [23] considers manually constructed formal models of vulnerability patterns to generate test cases for ANDROID applications. Test generation is implemented with an ad-hoc algorithm that also considers the compiled code of the application and the configuration files in the ANDROID system.

7 Conclusions and Future Work

ANDROID systems have a complex architecture designed to support the concurrent execution of applications on devices with limited resources. Here we have presented a model-based testing approach for generating test cases for ANDROID applications, which takes into account the way in which these applications interact with the user and with each other. We model the expected user behavior by composing state machines, and then explore this model exhaustively with SPIN to obtain all possible user behaviors, which correspond to test cases. These test cases are then executed in the device simulating the user inputs. In contrast with other approaches that generate random input events, our approach produces realistic user behaviors. Although our tool is currently geared towards ANDROID, the same principles can be applied to analyze applications in other mobile platforms, such as iOS and WINDOWS MOBILE.

The next step of our work will be to connect the generated test cases with a runtime verification monitor DRAGONFLY [14, 13]. In addition, we are working in adding more runtime information, like energy consumption, to perform richer analysis.

References


ioco theory for probabilistic automata

Marcus Gerhold
University of Twente, Enschede, The Netherlands
m.gerhold@utwente.nl

Mariëlle Stoelinga
University of Twente, Enschede, The Netherlands
marielle@cs.utwente.nl

Model-based testing (MBT) is an well-known technology, which allows for automatic test case generation, execution and evaluation. To test non-functional properties, a number of test MBT frameworks have been developed to test systems with real-time, continuous behaviour, symbolic data and quantitative system aspects. Notably, a lot of these frameworks are based on Tretmans’ classical input/output conformance (ioco) framework. However, a model-based test theory handling probabilistic behaviour does not exist yet. Probability plays a role in many different systems: unreliable communication channels, randomized algorithms and communication protocols, service level agreements pinning down up-time percentages, etc. Therefore, a probabilistic test theory is of great practical importance. We present the ingredients for a probabilistic variant of ioco and define the pioco relation, show that it conservatively extends ioco and define the concepts of test case, execution and evaluation.

1 Introduction

Model-based testing (MBT) is a way to test systems more effectively and more efficiently. By generating, executing and evaluating test cases automatically from a formal requirements model, more tests can be executed at a lower cost. A number of MBT tools have been developed, such as the Axini test manager, JTorx [1], STG [5], TorXakis [18], Uppaal-Tron [10, 16], etc.

A wide variety of model-based test theories exist: the seminal theory of Input/Output conformance [25, 27] is able to test functional properties, and has established itself as the robust core with a wide number of extensions. The correct functioning of today’s complex cyberphysical systems, depends not only on functional behaviour, but largely on non-functional, quantitative system aspects, such as real-time and performance. MBT frameworks have been developed to support these aspects: To test timing requirements, such as deadlines, a number of timed ioco-variants have been developed, such as [2, 10, 15]. Symbolic data can be handled by the frameworks in [8, 14]; resources by [3], and hybrid aspects in [19].

This paper introduces pioco, a conservative extension of ioco that is able to handle discrete probabilities. Starting point is a requirements model as a probabilistic quiescent transition system (pQTS), an input/output transition system, with two additional features: (1) Quiescence, which models the absence of outputs explicitly via a distinct δ label: quiescence is an important notion in ioco, because a system-under-test (SUT) may fail a certain test case given an output is required, but the SUT does not provide one. (2) Discrete probabilistic choice. We work in the input-generative / output-reactive model [9], which extend Segala’s classical probabilistic automaton model [20]: upon receiving an input, a pQTS chooses probabilistically, which target state to move to. For outputs, a pQTS chooses probabilistically both which action to take, and which state to move to, see Figure 1 for an example.

An important contribution of our paper is the notion of test case execution and evaluation. In particular, we show how the use of statistical hypothesis testing can be exploited to determine the verdict of a test execution: if we execute a test case sufficiently many times and the observed trace frequencies do not
coincide with the probabilities described in the specification pQTS depending on a predefined level of significance, then we fail the test case. In this way, we obtain a clean framework for test case generation, evaluation and execution. However, being a first step, we mainly establish the theoretical background. Further Research is needed to implement this theory into a working tool for probabilistic testing.

**Related work.** An early and influential paper on probabilistic testing is *Bisimulation Through Probabilistic Testing* [17], which not only defines the fundamental concept of probabilistic bisimulation, but also shows how different (i.e. non-bisimilar) probabilistic behaviours can be detected via statistical hypothesis testing. This idea has been taken further in our earlier work [4, 22], which shows how to observe trace probabilities via hypothesis testing.

Testing probabilistic Finite State Machines is well-studied (e.g. [13]) and coincidences to ioco theory can be found. However pQTS are more expressive than PFSMs, as they support non-determinism and underspecification, which both play a fundamental role in testing practice. Hence, they provide more suitable models for today’s highly concurrent and cyberphysical systems.

A paper that is similar in spirit to ours is by Hierons et al. [11, 12], and also considers input reactive / output generative systems with quiescence. However, there are a few important differences: Our model can be considered as an extension of [11] reconciling probabilistic and nondeterministic choices in a fully fledged way. Being more restrictive enables [11, 12] to focus on individual traces, whereas we use trace distributions.

Other work that involves the use of probability is given in [7, 28, 29], which models the behaviour of the tester, rather than of the SUT as we do, via probabilities.

**Organization of the paper.** We start by defining overall preliminaries in Section 2. Section 3 defines the conformance relation ioco for those systems and Section 4 provides the structure for testing and denotes what it means for an implementation to fail or pass a test suite by the means of an output and a statistical verdict. The paper ends with conclusions and future work in Section 5.

## 2 Probabilistic quiescent transition systems

### 2.1 Basic definitions

**Definition 1.** (Probability Distribution) A *discrete probability distribution* over a set $X$ is a function $\mu : X \rightarrow [0, 1]$ such that $\sum_{x \in X} \mu(x) = 1$. The set of all distributions over $X$ is denoted as $\text{Distr}(X)$. The probability distribution that assigns 1 to a certain element $x \in X$ is called the *Dirac* distribution over $x$ and is denoted $\text{Dirac}(x)$.

**Definition 2.** (Probability Space) A *probability space* is a triple $(\Omega, \mathcal{F}, P)$, such that $\Omega$ is a set, $\mathcal{F}$ is a $\sigma$-field of $\Omega$, and $P : \mathcal{F} \rightarrow [0, 1]$ a probability measure such that $P(\Omega) = 1$ and $P(\bigcup_{i=0}^\infty A_i) = \sum_{i=0}^\infty P(A_i)$ for $A_i$, $i = 1, 2, \ldots$ pairwise disjoint.

### 2.2 Probabilistic quiescent transition systems

As stated, we consider probabilistic transitions that are *input reactive* and *output generative* [9]: upon receiving an input, the system decides probabilistically which next state to move to. However, the system cannot decide probabilistically which inputs to accept. For outputs, in contrast, a system may make a probabilistic choice over various output actions. This means that each transition in a pQTS either involves...
a single input action, and a probabilistic choice over the target states; or it makes a probabilistic choice over several output actions, together with their target states. We refer to Figure 1 for an example.

Moreover, we model quiescence explicitly via a $\delta$-label. Quiescence means absence of outputs and is essential for testing: if the SUT does not provide any outputs, a test must determine whether or not this behaviour is correct. In the non-probabilistic case, this can be done either via the suspension automaton construction [26], or via QTSs [23]. The SA construction involves determinization. However, this is an ill-defined term for probabilistic systems. Therefore, we use the quiescent-labelling approach and demand to make quiescence explicit.

Finally, we assume that our pQTSs are finite and don’t contain internal steps (i.e., $\tau$-transitions).

**Definition 3.** (pQTS) A probabilistic quiescent transition system (pQTS) is an ordered five tuple $\mathcal{A} = (S, s_0, L_I, L_O, \Delta)$ where

- $S$ a finite set of states,
- $s_0 \in S$ the initial state,
- $L_I$ and $L_O^{\delta}$ disjoint sets of input and output actions, with at least $\delta \in L_O^{\delta}$. We write $L := L_I \cup L_O^{\delta}$ for the set of all labels and let $L_O = L_O^{\delta} \setminus \{\delta\}$ the set of all real outputs.
- $\Delta \subseteq S \times \text{Distr}(L \times S)$ a finite transition relation such that for all $(s, \mu) \in \Delta, a? \in L_I, b \in L, s', s'' \in S$, if $\mu(a?, s') > 0$, then $\mu(b, s'') = 0$ for all $b \neq a$.

We write $s \xrightarrow{\mu, a} s'$ if $(s, \mu) \in \Delta$ and $\mu(a, s') > 0$; and $s \rightarrow a$ if there are $\mu \in \text{Distr}(L \times S)$ and $s' \in S$ such that $s \xrightarrow{\mu, a} s'$. If it is not clear from the context about which system we are talking, we will write $s \xrightarrow{\mu, a} s'$, $(s, \mu)_{\mathcal{A}}$ and $s \rightarrow_a \mathcal{A}$ to clarify ambiguities. Lastly we say that $\mathcal{A}$ is input enabled if for all $s \in S$ we have $s \rightarrow_a \mathcal{A}$ for every $a \in L_I$.

### 2.3 Paths and traces

We define the usual language-theoretic concepts for pQTSs.

**Definition 4.** Let $\mathcal{A} = (S, s_0, L_I, L_O^{\delta}, \Delta)$ be a pQTS.

- A path $\pi$ of a pQTS $\mathcal{A}$ is a (possibly) infinite sequence of the form
  $$\pi = s_1 \mu_1 a_1 s_2 \mu_2 a_2 s_3 \mu_3 a_3 s_4 \ldots,$$
  where $s_i \in S$, $a_i \in L$ for $i = 1, 2, \ldots$ and $\mu \in \text{Distr}(L, S)$, such that each finite path ends in a state and $s_i \xrightarrow{\mu_i, a_i} s_{i+1}$ for each nonfinal $i$. We use the notation $\text{first}(\pi) := s_1$ to denote the first state of a path, as well as $\text{last}(\pi) := s_n$ for a finite path ending in $s_n$, and $\text{last}(\pi) = \infty$ for infinite paths. The set of all finite paths of a pQTS $\mathcal{A}$ is denoted by $\text{Path}^f(\mathcal{A})$ and the set of all infinite paths by $\text{Path}(\mathcal{A})$ respectively.

- The trace of a path $\pi = s_1 \mu_1 a_1 s_2 \mu_2 a_2 s_3 \ldots$ is the sequence obtained by omitting everything but the action labels, i.e. $\text{trace}(\pi) = a_1 a_2 a_3 \ldots$.

- All finite traces of $\mathcal{A}$ are summarized in $\text{traces}(\mathcal{A}) = \{\text{trace}(\pi) \in L^* \mid \pi \in \text{Path}^f(\mathcal{A})\}$.

- We write $s_1 \xrightarrow{\sigma} s_n$ with $\sigma \in L^*$ for $s_1, s_n \in S$ in case there is a path $\pi = s_1 \mu_1 a_1 \ldots \mu_{n-1} a_{n-1} s_n$ with $\text{trace}(\pi) = \sigma$ and $s_i \xrightarrow{\mu_i, a_i} s_{i+1}$ for $i = 1, \ldots, n - 1$.

- We write $\text{reach}_{\mathcal{A}}(S', \sigma)$ for the set of reachable states of a subset $S' \subseteq S$ via $\sigma$, i.e.
  $$\text{reach}_{\mathcal{A}}(S', \sigma) = \{s \in S \mid \exists s' \in S' : s' \xrightarrow{\sigma} s\}.$$
• All complete initial traces of $\mathcal{A}$ are denoted by $\text{ctraces} (\mathcal{A})$, which is defined as the set

$$\{\text{trace} (\pi) \mid \pi \in \text{Path} (\mathcal{A}) : \text{first} (\pi) = s_0, |\pi| = \infty \lor \forall a \in L : \text{reach} (\mathcal{A}) (\text{last} (\pi), a) = \emptyset\}.$$ 

• We write $\text{after}_\mathcal{A} (s)$ for the set of actions, enabled from state $s$, i.e. $\text{after}_\mathcal{A} (s) = \{a \in L \mid s \rightarrow a\}$. We lift this definition to traces by defining

$$\text{after}_\mathcal{A} (\sigma) = \bigcup_{s \in \text{reach}_\mathcal{A} (s_0, \sigma)} \text{after}_\mathcal{A} (s).$$ 

• We write $\text{out}_\mathcal{A} (\sigma) = \text{after}_\mathcal{A} (\sigma) \cap L^\delta_O$ to denote the set of all output actions as well as quiescence after trace $\sigma$.

In order for a pQTS to be meaningful, [23] postulated four well-formedness rules about quiescence, stating for instance that quiescence should not be succeeded by an output action. Since our current treatment does not rely on well-formedness, we omit these rules here. Moreover, our definition of a test case is a pQTS that does not adhere to the well-formedness criteria.

### 2.4 Trace distributions

Very much like the visible behaviour of a labelled transition system is given by its traces, the visible behaviour of a pQTS is given by its trace distributions: each trace distribution is a probability space that assigns a probability to (sets of) traces [20]. Just as a trace in an LTS is obtained by first selecting a path in the LTS and by then removing all states and internal actions, we do the same in the probabilistic case: we first resolve all the nondeterministic choices in the pQTS via an adversary, and by then removing all states — recall that our pQTSs do not contain internal actions. The resolution of the nondeterminism via an adversary leads to a purely probabilistic structure where we can assign a probability to each finite path, by multiplying the probabilities along that path. The mathematics to handle infinite paths is more complex, but completely standard [6]: in non-trivial situations, the probability assigned to an individual trace is 0 (cf., the probability to always roll a 6 with a dice is 0). Hence, we consider the probability assigned to sets of traces (e.g., the probability that a 6 turns up in the first 100 dice rolls). A classical result in measure theory shows that it is impossible to assign a probability to all sets of traces. Therefore, we collect those sets that can be assigned a probability in a so-called $\sigma$-field $\mathcal{F}$.

**Adversaries.** Following the standard theory for probabilistic automata [21], we define the behavior of a pQTS via adversaries (a.k.a. policies or schedulers). These resolve the nondeterministic choices in pQTSs: in each state of the pQTSs, the adversary chooses which transition to take. Adversaries can be (1) history-dependent, i.e. the choice which transition to take can depend on the full history; (2) randomized, i.e. the adversary may make a random choice over all outgoing transitions; and (3) partial, i.e., at any point in time, a scheduler may decide, with some probability, to terminate the execution.

Thus, given any finite history leading to a current state, an adversary returns a discrete probability distribution over the set of available next transitions (distributions to be precise). In order to model termination, we define schedulers which continue the transitions of pQTSs with a halting extension.

**Definition 5.** (Adversary) A (partial, randomized, history-dependent) adversary $E$ of a pQTS $\mathcal{A} = (S, s_0, L^I, L^O, \Delta)$ is a function

$$E : \text{Path}^\ast (\mathcal{A}) \longrightarrow \text{Distr} (\text{Distr} (L \times S) \cup \{\bot\}) ,$$
such that for each finite path \( \pi \), if \( E(\pi)(\mu) > 0 \), then \((\text{last} (\pi), \mu) \in \Delta\). The value \( E(\pi)(\bot) \) is considered as interruption/halting. We say that \( E \) is deterministic, if \( E(\pi) \) assigns the Dirac distribution for every distribution after all \( \pi \in \text{Path}^* (\mathcal{A}) \). An adversary \( E \) halts on a path \( \pi \), if it extends \( \pi \) to the halting state \( \bot \), i.e.

\[
E(\pi)(\bot) = 1.
\]

We say that an adversary halts after \( k \in \mathbb{N} \) steps, if it halts for every path \( \pi \) with \( |\pi| \geq k \). We denote all such adversaries by \( \text{Adv}(\mathcal{A}, k) \). Lastly \( E \) is finite, if there exists \( k \in \mathbb{N} \) such that \( E \in \text{Adv}(\mathcal{A}, k) \).

**The probability space assigned to an adversary.** Intuitively an adversary tosses a coin at every step of the computation, thus resulting in a purely probabilistic (as opposed to nondeterministic) computation tree.

**Definition 6.** (Path Probability) Let \( E \) be an adversary of \( \mathcal{A} \). The function \( Q^E : \text{Path}^* (\mathcal{A}) \to [0, 1] \) is called the path probability function and it is defined by induction. We set \( Q^E(s_0) = 1 \) and \( Q^E(\pi \mu a s) = Q^E(\pi) \cdot E(\pi)(\mu) \cdot \mu(a, s) \).

Loosely speaking, we follow a finite path in the transition system and multiply every scheduled probability along the way, resolving every nondeterminism according to the adversary \( E \) to get the ultimate path probability. The path probability function enables us to define a probability space associated with an adversary, thus giving every path in a pQTS \( \mathcal{A} \) an exact probability.

**Definition 7.** (Adversary Probability Space) Let \( E \) be an adversary of \( \mathcal{A} \). The unique probability space associated to \( E \) is the probability space \( (\Omega_E, \mathcal{F}_E, P_E) \) given by.

1. \( \Omega_E = \text{Path}^\infty (\mathcal{A}) \)
2. \( \mathcal{F}_E \) is the smallest \( \sigma \)-field that contains the set \( \{ C_\pi \mid \pi \in \text{Path}^* (\mathcal{A}) \} \), where the cone is defined as \( C_\pi = \{ \pi' \in \Omega_E \mid \pi \text{ is a prefix of } \pi' \} \).
3. \( P_E \) is the unique probability measure on \( \mathcal{F}_E \) s. t. \( P_E(C_\pi) = Q^E(\pi) \), for all \( \pi \in \text{Path}^* (\mathcal{A}) \).

The set of all adversaries is denoted by \( \text{adv}(\mathcal{A}) \) with \( \text{adv}(\mathcal{A}, k) \) being the set of adversaries halting after \( k \in \mathbb{N} \) respectively.

**Trace distributions.** As we mentioned, a trace distribution is obtained from (the probability space assigned to) an adversary by removing all states. This means that the probability assigned to a set of traces \( X \) is defined as the probability of all paths whose trace is an element of \( X \).

**Definition 8.** (Trace Distribution) The trace distribution \( H \) of an adversary \( E \), denoted \( H = \text{trd}(E) \) is the probability space \( (\Omega_H, \mathcal{F}_H, P_H) \) given by

1. \( \Omega_H = L^*_{\mathcal{A}'} \)
2. \( \mathcal{F}_H \) is the smallest \( \sigma \)-field containing the set \( \{ C_\beta \mid \beta \in L^*_{\mathcal{A}'} \} \), where the cone is defined as \( C_\beta = \{ \beta' \in \Omega_E \mid \beta \text{ is a prefix of } \beta' \} \)
3. \( P_H \) is the unique probability measure on \( \mathcal{F}_H \) such that \( P_H(X) = P_E(\text{trace}^{-1}(X)) \) for \( X \in \mathcal{F}_H \).

As an abbreviation, we will write \( P_H(\beta) := P_H(C_\beta) \) for \( \beta \in L^*_{\mathcal{A}'} \).

Like before, we denote the set of trace distributions based on adversaries of \( \mathcal{A} \) by \( \text{trd}(\mathcal{A}) \) and \( \text{trd}(\mathcal{A}, k) \) if it is based on an adversary halting after \( k \in \mathbb{N} \) steps respectively. Lastly we write \( \mathcal{A} =_{TD} \mathcal{B} \) if \( \text{trd}(\mathcal{A}) = \text{trd}(\mathcal{B}) \), \( \mathcal{A} \subseteq_{TD} \mathcal{B} \) if \( \text{trd}(\mathcal{A}) \subseteq \text{trd}(\mathcal{B}) \) and \( \mathcal{A} \subseteq_{TD}^k \mathcal{B} \) if \( \text{trd}(\mathcal{A}, k) \subseteq \text{trd}(\mathcal{B}, k) \) for \( k \in \mathbb{N} \).
where the embedding means that for every trace distribution $H$ of $\mathcal{A}$ there is a trace distribution $H'$ of $\mathcal{B}$ such that for all traces $\sigma$ of $\mathcal{A}$, we have $P_H(\sigma) = P_{H'}(\sigma)$.

The fact that $(\Omega_E, \mathcal{F}_E, P_E), (\Omega_H, \mathcal{F}_H, P_H)$ really define probability spaces, follows from standard measure theory arguments (see [6]).

**Example 9.** Consider the pQTS $\mathcal{A} = (S, s_0, L_I, L^B_0, \Delta)$ in Figure 1. There $S = \{s_0, s_1, \ldots, s_{10}\}$, $L_I = \{a?\}$, $L^B_0 = \{b!, c!, d!\} \cup \{\delta\}$ and $\Delta = \{(s_0, \mu_{0_1}), (s_2, \mu_{0_2}), (s_0, \mu_{0_3}), (s_1, \mu_1), \ldots, (s_{10}, \mu_{10})\}$. We can see that this system has both probabilistic and nondeterministic choices. Observe that it has indeed only input reactive and output generative transitions as mentioned in the beginning of 2.2.

We will now consider an adversary $E$ for $\mathcal{A}$. The only nondeterministic choice we have in this system, is located at state $s_0$, where we can either apply $a?$ to enter the left branch, $a?$ to enter the right branch, or do nothing (corresponding to $\mu_{0_1}$, $\mu_{0_2}$ and $\mu_{0_3}$, respectively). Therefore consider the adversary $E (s_0) (\mu_{0_1}) = \frac{1}{3}$ and $E (s_0) (\mu_{0_2}) = \frac{1}{3}$ and $E (\pi) (\mu) = Dirac$ for every other distribution $\mu$ after a path $\pi$ (i.e. those are taken with probability 1).

The adversary probability space created for this adversary assigns an unambiguous path probability to each path. Consider the path $\pi = s_0\mu_{0_1}a?s_1\mu_1b!s_5$, then

$$P_E(\pi) = Q^E(\pi) = \frac{Q^E(s_0)}{1} (\mu_{0_1}) \frac{Q^E(s_0)}{1} (\mu_{0_1}) (a?,s_1) (\mu_1) \frac{Q^E(s_0)}{1} (\mu_{0_1}) (a?,s_1) (\mu_1) (b!,s_5) = \frac{1}{8}. $$

However, consider the trace distribution $H = trd(E)$. Then for $\sigma = a?b!$, we have $\text{trace}^{-1}(\sigma) = \{\pi, \eta\}$ with $\eta = s_0\mu_{0_2}a?s_3\mu_2b!s_8$. Hence

$$P_H(\sigma) = P_{\text{trd}(E)}(\text{trace}^{-1}(\sigma)) = P_E(\{\pi, \eta\}) = P_E(\pi) + P_E(\eta) = \frac{1}{4}.$$

### 3 The probabilistic conformance relation $\pioco$

#### 3.1 The $\pioco$ relation

The classical input-output conformance relation $ioco$ states that an implementation $\mathcal{A}$ conforms to a specification $\mathcal{A}$, if $\mathcal{A}$ never provides any unspecified output. In particular this refers to the observation of quiescence, when other output was expected.
Definition 10. (Input-Output Conformance) Let $\mathcal{A}_1$ and $\mathcal{A}_s$ be two QTS and let $\mathcal{A}_i$ be input enabled. Then we say $\mathcal{A}_1 \sqsubseteq_{\text{ioco}} \mathcal{A}_s$, if and only if

$$\forall \sigma \in \text{traces}(\mathcal{A}_s) : \text{out}_{\mathcal{A}_1}(\sigma) \subseteq \text{out}_{\mathcal{A}_i}(\sigma).$$

To generalize ioco to pQTSs, we introduce two auxiliary concepts. For a natural number $k$, the prefix relation $H \sqsubseteq_k H'$ states that trace distribution $H$ assigns exactly the same probabilities as $H'$ to traces of length $k$ and halts afterwards. The output continuation of a trace distribution $H$ prolongs the traces of $H$ with output actions. More precisely, output continuation of $H$ wrt length $k$ contains all trace distributions that (1) coincide with $H$ for traces up to length $k$ and (2) the $k+1$st action is an output label ( incl $\delta$); i.e. traces of length $k+1$ that end on an input action are assigned probability 0. Recall that $P_H(\sigma)$ abbreviates $P_H(C_\sigma)$.

Definition 11. (Notations) For a natural number $k \in \mathbb{N}$, and trace distributions $H \in \text{trd}(\mathcal{A}, k)$, we say that

1. $H$ is a prefix of $H' \in \text{trd}(\mathcal{A})$ up to $k$, denoted by $H \sqsubseteq_k H'$, if $\forall \sigma \in L^k : P_H(\sigma) = P_{H'}(\sigma)$.
2. the output continuation of $H$ in $\mathcal{A}$ is given by

$$\text{out}_{\mathcal{A}}(H, k) := \left\{ H' \in \text{trd}(\mathcal{A}, k+1) \mid H \sqsubseteq_k H' \land \forall \sigma \in L^k L^\delta : P_{H'}(\sigma) = 0 \right\}.$$

We are now able to define the core idea of ioco. Intuitively an implementation should conform to a specification, if the probability of every trace in $\mathcal{A}_1$ specified in $\mathcal{A}_s$, can be matched in the specification. Just as in ioco, we will neglect underspecified traces continued with input actions (i.e., everything is allowed to happen after that). However, if there is unspecified output in the implementation, there is at least one adversary that schedules positive probability to this continuation, which consequently cannot be matched of output-continuations in the specification.

Definition 12. Let $\mathcal{A}_i$ and $\mathcal{A}_s$ be two pQTS. Furthermore let $\mathcal{A}_i$ be input enabled, then we say $\mathcal{A}_1 \sqsubseteq_{\text{pioco}} \mathcal{A}_s$ if and only if

$$\forall k \in \mathbb{N} \forall H \in \text{trd}(\mathcal{A}_s, k) : \text{out}_{\mathcal{A}_1}(H, k) \subseteq \text{out}_{\mathcal{A}_s}(H, k).$$

Example 13. Consider the two systems of $\mathcal{A}$ and $\mathcal{B}$ shown in Figure 2 and assume that $p \in [0, 1]$. It is true that $\mathcal{A} \sqsubseteq_{\text{pioco}} \mathcal{B}$, because we can always choose an adversary $E$ of $\mathcal{B}$, which imitates the probabilistic behaviour of $\mathcal{B}$, i.e. choose $E(\varepsilon)(\mu) = \nu$ such that $\nu(a!, t_1) = p$ and $\nu(b!, t_2) = 1 - p$.

However, the opposite does not hold. For example assume $p = \frac{1}{2}$, then the trace distribution $H$ assigning $P_H(a!) = 1$ is in $\text{out}_{\mathcal{B}}(H, 1)$ but not in $\text{out}_{\mathcal{A}}(H, 1)$ and hence $\mathcal{B} \not\sqsubseteq_{\text{pioco}} \mathcal{A}$.
3.2 Properties of the p-ioco relation

As stated before, the relation pioco conservatively extends the ioco relation, i.e. both relations coincide for non-probabilistic QTSs. Moreover, we show that several other characteristic properties of ioco carry over to pioco as well. Below, a QTS is a pQTS where every occurring distribution is the Dirac distribution.

**Theorem 14.** Let $\mathcal{A}_i$ and $\mathcal{A}_s$ be two QTS and let $\mathcal{A}_i$ be input enabled, then

$$\mathcal{A}_i \subseteq_{ioco} \mathcal{A}_s \iff \mathcal{A}_i \subseteq_{pioco} \mathcal{A}_s.$$  

Intuitively it makes sense that the implementation is input enabled, since it should accept every input at any time. The following two results justify, that we assume the specification to be not input enabled, since otherwise pioco would coincide with trace distribution inclusion. Equivalently it is known that ioco coincides with trace inclusion, if we assume both the implementation and the specification were input enabled. Thus, as stated before, we can see that pioco extends ioco.

**Lemma 15.** Let $\mathcal{A}_i$ and $\mathcal{A}_s$ be two pQTS, then

$$\mathcal{A}_i \subseteq_{TD} \mathcal{A}_s \implies \mathcal{A}_i \subseteq_{pioco} \mathcal{A}_s.$$  

**Theorem 16.** Let $\mathcal{A}_i$ and $\mathcal{A}_s$ be two input enabled pQTS, then

$$\mathcal{A}_i \subseteq_{pioco} \mathcal{A}_s \iff \mathcal{A}_i \subseteq_{TD} \mathcal{A}_s.$$  

Next, we show that, under some input-enabledness restrictions, the pioco relation is transitive. Again, note that this is also true for ioco for non-probabilistic systems.

**Theorem 17.** (Transitivity of pioco) Let $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$ be pQTS, such that $\mathcal{A}$ and $\mathcal{B}$ are input enabled, then

$$\mathcal{A} \subseteq_{pioco} \mathcal{B} \land \mathcal{B} \subseteq_{pioco} \mathcal{C} \implies \mathcal{A} \subseteq_{pioco} \mathcal{C}.$$  

4 Testing for pQTS

4.1 Test cases for pQTSs.

We will consider tests as sets of traces based on an action signature $(L_I, L_O, \delta)$, which will describe possible behaviour of the tester. This means that at each state in a test case, the tester either provides stimuli or waits for a response of the system. Additionally to output conformance testing like in [24], we introduce probabilities into our testing transition system. Thus we can represent each test case as a pQTS, albeit with a mirrored action signature $(L_O, L_I \cup \{\delta\})$. This is necessary for the parallel composition of the test pQTS and the SUT.

Since we consider tests to be pQTS, we also use all the terminology introduced earlier on. Additionally we require tests to not contain loops (or infinite paths respectively).

**Definition 18.** A test (directed acyclic graph) over an action signature $(L_I, L_O, \delta)$ is a pQTS of the form $t = (S, s_0, L_O, L_I \cup \{\delta\}, \Delta)$ such that

- $t$ is internally deterministic and does not contain an infinite path;
- $t$ is acyclic and connected;
• For every state $s \in S$, we either have
  - $\text{after}(s) = \emptyset$, or
  - $\text{after}(s) = L_I \cup \{\delta\}$, or
  - $\text{after}(s) = \{a!\} \cup L_I \cup \{\delta\}$ for some $a! \in L_O$.

A test suite $T$ is a set of tests over an action signature $(L_I, L_O, L^\delta_O)$. We write $T(L_I, L_O, L^\delta_O)$ to denote all the tests over an action signature $(L_I, L_O, L^\delta_O)$ and $T S(L_I, L_O, L^\delta_O)$ as the set of all test suites over an action signature respectively.

For a given specification pQTS $\mathcal{A}_s = (S, s_0, L_I, L^\delta_O, \Delta)$, we say that a test $t$ is a test for $\mathcal{A}_s$, if it is based on the same action signature $(L_I, L^\delta_O, L^\delta_O)$. Similar to before, we denote all tests for $\mathcal{A}_s$ by $T(\mathcal{A}_s)$ and all test suites by $T S(\mathcal{A}_s)$ respectively.

Note that we mirrored the action signature for tests, as can be seen in Figure 3a and Figure 3b respectively. That is, because we require tests and implementations to shake hands on shared actions. A special role is dedicated to quiescence in the context of parallel composition, since the composed system is considered quiescent if and only if the two systems are quiescent.

We will proceed to define parallel composition. Formally this means that output actions of one component are allowed to be present as input actions of the other component. These will be synchronized upon. However, keeping in mind the mirrored action signature of tests, we wish to avoid possibly unwanted synchronization, which is why we introduce system compatibility.

**Definition 19.** (Compatibility) Two pQTS $\mathcal{A} = (S, s_0, L_I, L^\delta_O, \Delta)$, and $\mathcal{A}' = (S', s'_0, L'_I, L'^\delta_O, \Delta')$ are said to be compatible if $L^\delta_O \cap L'^\delta_O = \{\delta\}$.

When we put two pQTSs in parallel, they synchronize on shared actions, and evolve independently on others. Since the transitions taken by the two component of the composition are stochastically independent, we multiply the probabilities when taking shared actions.

**Definition 20.** (Parallel composition) Given two compatible pQTS $\mathcal{A} = (S, s_0, L_I, L^\delta_O, \Delta)$ and $\mathcal{A}' = (S', s'_0, L'_I, L'^\delta_O, \Delta')$, their parallel composition is the tuple $\mathcal{A} || \mathcal{A}' = (S'', s''_0, L''_I, L''^\delta_O, \Delta'')$, where $S'' = S \times S'$.
\[ s_0'' = (s_0, s_0'), \]
\[ L''_0 = (L_I \cup L'_I) \setminus (L_O \cup L'_O), \]
\[ L''_O = L''_O \cup L''_O', \]
\[ \Delta'' = \{(s,t), \mu) \in S'' \times \text{Distr}(L'' \times S'') \} \]

\[
\mu(a, (s', t')) = \begin{cases} 
\mu_a(a, s') \nu_a(a, t') & \text{if } a \in L \cap L', \text{ where } s \xrightarrow{\mu_a, a} s' \wedge t \xrightarrow{\nu_a, a} t' \\
\mu_a(a, s') & \text{if } a \in L \setminus L', \text{ where } s \xrightarrow{\mu_a, a} s' \wedge t = t' \\
\nu_a(a, t') & \text{if } a \in L' \setminus L, \text{ where } s = s' \wedge t \xrightarrow{\nu_a, a} t' \\
0 & \text{otherwise}
\end{cases}
\]

where \( \mu_a \in \text{Distr}(L, S) \) and \( \nu_a \in \text{Distr}(L', S') \) respectively.

Before we parallel compose a test case with a system, we obviously need to define which outcome of a test case is considered correct, and which is not (i.e., when it fails).

**Definition 21.** (Test case annotation) For a given test \( t \) a test annotation is a function

\[ a : \text{traces}(t) \longrightarrow \{\text{pass, fail}\}. \]

A pair \( \hat{t} = (t, a) \) consisting of a test and a test annotation is called an annotated test. The set of all such \( \hat{t} \) is defined as \( \hat{T} = \{(t_i, a_i)_{i \in \mathcal{I}}\} \) for some index set \( \mathcal{I} \) is called an annotated Test Suite. If \( t \) is a test case for a specification \( \mathcal{A} \), we define the pioco test annotation \( a^{\text{pioco}}_{\mathcal{A}, \hat{t}} : \text{traces}(t) \longrightarrow \{\text{pass, fail}\} \) by

\[
a^{\text{pioco}}_{\mathcal{A}, \hat{t}}(\sigma) = \begin{cases} 
\text{fail} & \text{if } \exists \sigma_1 \in \text{traces}(\mathcal{A}), a! \in L''_O : \sigma_1 a! \sqsubseteq \sigma \wedge \sigma_1 a! \notin \text{traces}(\mathcal{A}); \\
\text{pass} & \text{otherwise.}
\end{cases}
\]

### 4.2 Test execution.

By taking the intersection of all complete traces within a test and all traces of an implementation, we will define the set of all traces that will be executed by an annotated test case.

**Definition 22.** (Test execution) Let \( t \) be a test over the action signature \( (L_I, L''_O) \) and the pQTS \( \mathcal{A}_i = (S, s_0, L_I, L''_O, \Delta) \). Then we define

\[ \text{exec}_t(\mathcal{A}_i) = \text{traces}(\mathcal{A}_i) \cap \text{traces}(\hat{t}). \]

**Example 23.** Consider the specification of a shuffle music player and a derived test for it given in Figure 3. Assuming we are to test whether or not the following two implementations conform to the specification with respect to pioco:
Here \( p_1, \ldots, p_N \in [0, 1] \) such that \( \sum_{i=1}^{N} p_i = 1 \). Now when we compose \( \omega_i \) with \( \tau \) in Figure 3b, we can clearly see that every complete trace of the parallel system is annotated with \( \text{fail} \), as it would also have been the case for classical ioco theory. However, if we now also consider \( \omega_i \) and compose it with the same test \( \tau \), every trace of the composed system would be given a \( \text{pass} \) label if we restricted ourselves to the annotation function and the output verdict. Note how every trace \( \text{shuffle?}. \text{Song} \) is given probability \( p_i \) for \( i = 1, \ldots, N \). The only restriction we assumed valid for \( p_1, \ldots, p_N \) is that they sum up to 1 so a correct distribution for \( \omega_i \) would be \( p_1 = \frac{N-1}{N} \) and \( p_2 = \ldots = p_N = \frac{1}{N^2} \). This, however, should intuitively not be given the verdict \( \text{pass} \), since it differs from the uniform distribution given in the specification \( \omega_i \).

### 4.3 Test evaluation

In order to give a verdict of whether or not the implementation passed the test (suite), we need to extend the test evaluation process of classical ioco testing with a statistical component. Thus the idea of evaluating probabilistic systems becomes two folded. On the one hand, we want that no unexpected output (or unexpected quiescence) ever occurs during the execution. On the other hand, we want the observed frequencies of the SUT to conform in some way to the probabilities described in the specification. Thus the SUT will pass the test suite only if it passes both criteria. We will do this by augmenting classical ioco theory with zero hypothesis testing, which will be discussed in the following.

To conduct an experiment, we need to define a length \( k \in \mathbb{N} \) and a width \( m \in \mathbb{N} \) first. This refers to how long the traces we want to record should be and how many times we reset the machine. This will give us traces \( \sigma_1, \ldots, \sigma_m \in L^k \), which we call a \emph{sample}. Additionally, we assume that the implementation is governed by an underlying trace distribution \( H \) in every run, thus running the machine \( m \) times, gives us a sequence of possibly \( m \) different trace distributions \( \bar{H} = H_1, \ldots, H_m \). So in every run the implementation makes two choices: 1) It chooses the trace distribution \( H \) and 2) \( H \) chooses a trace \( \sigma \) to execute. Consequently that means that once a trace distribution \( H_i \) is chosen, it is solely responsible for the trace \( \sigma_i \). Thus for \( i \neq j \) the choice of \( \sigma_i \) is independent from the choice of \( \sigma_j \).

Our statistical analysis is build upon the frequencies of traces occurring in a sample \( O \). Thus the \emph{frequency function} will be defined as

\[
\text{freq}(O)(\sigma) = \frac{|\{i \in \{1, \ldots, m\} \mid \sigma_i = \sigma\}|}{m}.
\]

Note that although every run is governed by possibly different trace distributions, we can still derive useful information from the frequency function. For fixed \( k, m \in \mathbb{N} \) and \( \bar{H} \), the sample \( O \) can be treated as a Bernoulli experiment of length \( m \), where success occurs in position \( i = 1, \ldots, m \), if \( \sigma = \sigma_i \). The success probability is then given by \( P_{H_i}(\sigma) \). So for given \( \bar{H} \), the expected value for \( \sigma \) is given by \( \mathbb{E}_{H_i}(\sigma) = \frac{1}{m} \sum_{i=1}^{m} P_{H_i}(\sigma) \). Note that this expected value \( \mathbb{E}_{\bar{H}} \) is the expected distribution over \( L^k \) if we assume it is based on the \( m \) trace distributions \( \bar{H} \).

In order to apply zero hypothesis testing and compare an observed distribution with \( \mathbb{E}_{\bar{H}} \), we will use the notion of metric spaces. This will enable us to measure deviation of two distributions. We will use the metric space \( (L^k, \text{dist}) \), where \( \text{dist} \) is the euclidean distance of two distributions defined as

\[
\text{dist}(\mu, \nu) = \sqrt{\sum_{\sigma \in L^k} |\mu(\sigma) - \nu(\sigma)|^2}.
\]

Now that we have a measure of deviation, we can say that a sample \( O \) is accepted if \( \text{freq}(O) \) lies in some distance \( r \) of the expected value \( \mathbb{E}_{\bar{H}} \), or equivalently if \( \text{freq}(O) \) is contained in the closed ball \( B_r(\mathbb{E}_{\bar{H}}) = \{ \nu \in \text{Distr}(L^k) \mid \text{dist}(\nu, \mathbb{E}_{\bar{H}}) \leq r \} \). Then the set \( \text{freq}^{-1}(B_r(\mathbb{E}_{\bar{H}})) \) summarizes all samples that deviate at most \( r \) from the expected value.
An inherent problem of hypothesis testing are the type 1 and type 2 errors, i.e., the probability of falsely accepting the hypothesis or falsely rejecting it. This problem is established in our framework by the choice of a level of significance \( \alpha \in [0, 1] \) and connected with it, the choice of radius \( r \) for the ball mentioned above. So for a given level of significance \( \alpha \) the following choice of the radius will in some sense minimize the probability of false acceptance of an erroneous sample and of false rejection of a valid sample (i.e., at most \( \alpha \)).

\[
\bar{r} := \inf \left\{ r \mid P_{\bar{H}} \left( \text{freq}^{-1} \left( B_r \left( \mathbb{E}_{\bar{H}} \right) \right) \right) > 1 - \alpha \right\}.
\]

Thus assuming we have \( m \) different underlying trace distributions, we can determine when an observed sample seems reasonable and is declared valid. Unifying over all sets of such \( \bar{H} \), we will define the total set of acceptable outcomes, called Observations.

**Definition 24.** The acceptable outcomes of \( \bar{H} \) with significance level \( \alpha \in [0, 1] \) are given by the set of samples of length \( k \in \mathbb{N} \) and width \( m \in \mathbb{N} \), defined as

\[
\text{Obs} \left( \bar{H}, \alpha \right) := \text{freq}^{-1} \left( B_r \left( \mathbb{E}_{\bar{H}} \right) \right) = \left\{ O \in \left( L^k \right)^m \mid \text{dist} \left( \text{freq} \left( O \right), \mathbb{E}_{\bar{H}} \right) \leq \bar{r} \right\}.
\]

The set of observations of \( \mathcal{A} \) with significance level \( \alpha \in [0, 1] \) is given by

\[
\text{Obs} \left( \mathcal{A}, \alpha \right) = \bigcup_{\bar{H} \in \text{trd}(\mathcal{A}, k)^m} \text{Obs} \left( \bar{H}, \alpha \right).
\]

**Example 25.** Assume that the wanted level of significance is given by \( \alpha = 0.05 \) and consider the probabilistic automaton in Figure 4 representing the toss of a fair coin. Furthermore assume that we are given two samples of depth \( k = 2 \) and width \( m = 100 \).

To sample this case, assume \( E \) is the adversary that assigns probability equal to 1 to the unique outgoing transition (if there is one) and probability 1 to halting, in case there is no outgoing transition. We take \( H = \text{trd} \left( E \right) \) and can see, that then \( \mu_H \left( a?b! \right) = \mu_H \left( a?c! \right) = \frac{1}{2} \) and \( \mu_H \left( \sigma \right) = 0 \) for all other sequences \( \sigma \). We define \( H^{100} = (H_1, \ldots, H_{100}) \), where \( H_1 = \ldots = H_{100} = H \). As we can see, we have \( \mathbb{E}_{H^{100}} = \mu_H \). Since \( \mu_H \) only assigns positive probability to \( a?b! \) and \( a?c! \), we get \( P_{H^{100}} \left( B_r \left( \mu_H \right) \right) = \left( \left\{ O \mid \frac{1}{2} - r \leq \text{freq} \left( O \right) \left( a?b! \right) \leq \frac{1}{2} + r \right\} \right) \). One can show that the smallest ball, where this probability is greater or equal than 0.95 is given by the ball of radius \( \bar{r} = \frac{1}{10} \).

Thus a sample \( O_1 \), which consists of 42 times \( a?b! \) and 58 times \( a?c! \) is an observation, and a sample \( O_2 \), which consists of 38 times \( a?b! \) and 62 times \( a?c! \) is not.

Thus we can finally define a verdict function, that assigns pass when a test case never finds erroneous behaviour (i.e., wrong output or wrong probabilistic behaviour).
Definition 26. (Output verdict) Let \((L_I, L^\delta_O)\) be an action signature and \(\hat{t} = (t, a)\) an annotated test case over \((L_I, L^\delta_O)\). The output verdict function for \(\hat{t}\) is the function \(v_{\hat{t}}: pQTS \rightarrow \{\text{pass, fail}\}\), given for any pQTS \(\mathcal{A}\)

\[
v_{\hat{t}}(\mathcal{A}) = \begin{cases} 
  \text{pass} & \text{if } \forall \sigma \in \text{exec}_t(\mathcal{A}): a(\sigma) = \text{pass} \\
  \text{fail} & \text{otherwise}
\end{cases}
\]

(Statistical verdict) Additionally let \(\alpha \in [0, 1]\) and \(k, m \in \mathbb{N}\) and \(O \in \text{Obs}(\mathcal{A}||\hat{t}, \alpha) \subseteq (L^k)^m\), then the statistical verdict function is given by

\[
v^\alpha_{\hat{t}}(\mathcal{A}) = \begin{cases} 
  \text{pass} & \text{if } O \in \text{Obs}(\mathcal{A}, \alpha) \\
  \text{fail} & \text{otherwise}
\end{cases}
\]

(Verdict function) For any given \(\mathcal{A}\), we assign the verdict

\[
V^\alpha_{\hat{t}}(\mathcal{A}) = \begin{cases} 
  \text{pass} & \text{if } v_{\hat{t}}(\mathcal{A}) = v^\alpha_{\hat{t}}(\mathcal{A}) = \text{pass} \\
  \text{fail} & \text{otherwise}
\end{cases}
\]

We extend \(V^\alpha_{\hat{t}}\) to a function \(V^\alpha_{\hat{T}}: pQTS \rightarrow \{\text{pass, fail}\}\), which assigns verdicts to a pQTS based on a given annotated test suite by \(V^\alpha_{\hat{T}}(\mathcal{A}) = \text{pass}\) if for all \(\hat{t} \in \hat{T}\) and \(V^\alpha_{\hat{T}}(\mathcal{A}) = \text{fail}\) otherwise.

5 Conclusion and Future Work

We introduced the core of a probabilistic test theory by extending classical ioco theory. We defined the conformance relation pioco for probabilistic quiescent transition systems, and proved several characteristic properties. In particular, we showed that pioco is a conservative extension of ioco. Second, we have provided definitions of a test case, test execution and test evaluation. Here, test execution is crucial, since it needs to assess whether the observed behaviour respects the probabilities in the specification pQTS. Following [4], we have used statistical hypothesis testing here.

Being a first step, there is ample future work to be carried out. First, it is important to establish the correctness of the testing framework, by showing the soundness and completeness. Second, we would like to implement our framework in the MBT testing framework JTorX, and test realistic applications. Also, we would like to extend our theory to handle \(\tau\)-transitions. Finally, we think that tests themselves should be probabilistic, in particular since many MBT tools in practice do already choose their next action probabilistically.

References


ioco theory for probabilistic automata


Appendix

Below, we present the proofs of our theorems.

Proofs

Proof of Theorem 14.

"\(\Leftarrow\)\" Let \(\mathcal{A}_i \subseteq_{\text{pioco}} \mathcal{A}_s\) and \(\sigma \in \text{traces}(\mathcal{A}_s)\). Our goal is to show \(\text{out}_{\mathcal{A}_i}(\sigma) \subseteq \text{out}_{\mathcal{A}_s}(\sigma)\).

For \(\text{out}_{\mathcal{A}_i}(\sigma) = \emptyset\) we are done, since \(\emptyset \subseteq \text{out}_{\mathcal{A}_s}(\sigma)\) obviously.

So assume that there is \(b! \in \text{out}_{\mathcal{A}_i}(\sigma)\). We want to show that \(b! \in \text{out}_{\mathcal{A}_s}(\sigma)\). For this, let \(k = |\sigma|\) and \(H \in \text{trd}(\mathcal{A}_s,k)\) such that \(P_H(\sigma) = 1\), which is possible because \(\sigma \in \text{traces}(\mathcal{A}_s)\) and both \(\mathcal{A}_i\) and \(\mathcal{A}_s\) are non-probabilistic. The same argument gives us \(\text{outcont}(H,\mathcal{A}_s,k) \neq \emptyset\), because \(\sigma \in \text{traces}(\mathcal{A}_s)\).

Thus we have at least one \(H' \in \text{outcont}(H,\mathcal{A}_s,k)\) such that \(P_{H'}(\sigma b!) > 0\). Let \(\pi \in \text{trace}^{-1}(\sigma) \cap \text{Path}^*(\mathcal{A}_s)\). Now \(H' \in \text{outcont}(H,\mathcal{A}_s,k)\), because \(\mathcal{A}_i \subseteq_{\text{pioco}} \mathcal{A}_s\) by assumption and thus there must be at least one adversary \(E' \in \text{adv}(\mathcal{A}_s,k+1)\) such that \(\text{trd}(E') = H'\) and \(Q^E(\pi \cdot \text{Dirac} \cdot b!s') > 0\) for some \(s' \in S\). Hence \(E'(\pi)(\text{Dirac} \cdot b!,s') > 0\) and therefore with \(s' \in \text{reach}(\text{last}(\pi),b)\) this yields \(b! \in \text{out}_{\mathcal{A}_s}(\sigma)\).

"\(\Rightarrow\)\" Let \(\mathcal{A}_i \subseteq_{\text{pioco}} \mathcal{A}_s\), \(k \in \mathbb{N}\) and \(H^* \in \text{trd}(\mathcal{A}_s,k)\). Assume that \(H \in \text{outcont}(H^*,\mathcal{A}_i,k)\), then we want to show that \(H \in \text{outcont}(H^*,\mathcal{A}_s,k)\).

Therefore let \(E \in \text{adv}(\mathcal{A}_i,k+1)\) such that \(\text{trd}(E) = H\). If we can find \(E' \in \text{adv}(\mathcal{A}_s,k+1)\) such that \(\text{trd}(E) = \text{trd}(E')\), we are done. We will do this constructively in three steps.

1) By construction of \(H^*\) we know that there must be \(E' \in \text{adv}(\mathcal{A}_s,k+1)\), such that for all \(\sigma \in L_k\) we have \(P_{\text{trd}(E')}(\sigma) = P_{H^*}(\sigma) = P_{\text{trd}(E)}(\sigma)\). Thus \(H^* \subseteq_{\text{trd}} \text{trd}(E')\).

2) We did not specify the behaviour of \(E'\) for path of length \(k+1\). Therefore we choose \(E'\) such that for all traces \(\sigma \in L_k\) and \(a? \in L_1\) we have \(P_{\text{trd}(E')}(\sigma a?) = P_{\text{trd}(E)}(\sigma a?)\).

3) The last thing to show is that \(\text{trd}(E) = \text{trd}(E')\). Therefore let us now set the behaviour of \(E'\) for traces ending in outputs. Let \(\sigma \in \text{traces}(\mathcal{A}_i)\), then assume \(a! \in \text{out}_{\mathcal{A}_i}(\sigma)\) (if \(\text{out}_{\mathcal{A}_i}(\sigma) = \emptyset\) we are done immediately) and because \(\mathcal{A}_i \subseteq_{\text{pioco}} \mathcal{A}_s\), we know that \(a! \in \text{out}_{\mathcal{A}_s}(\sigma)\).

Now let \(p := P_{\text{trd}(E)}(\sigma) = P_{\text{trd}(E')}(\sigma)\) and \(q := P_{\text{trd}(E)}(\sigma a!)\). By equality of the trace distributions for traces up to length \(k\) we know that \(q \leq p \leq 1\) and therefore there is \(\alpha \in [0,1]\) such that \(q = p \cdot \alpha\). Let \(\text{traces}(\mathcal{A}_s) \cap \text{trace}^{-1}(\sigma) = \{\pi_1,\ldots,\pi_n\}\). Without loss of generality, we choose \(E'\) such that

\[
E'(\pi_i)(\text{Dirac}) = \begin{cases} 
\alpha & \text{if } i = 1 \\
0 & \text{else}
\end{cases}
\]

We constructed \(E' \in \text{adv}(\mathcal{A}_s,k+1)\), such that for all \(\sigma \in L^{k+1}\) we have \(P_{\text{trd}(E')}(\sigma) = P_{\text{trd}(E)}(\sigma)\) and thus \(\text{trd}(E) = \text{trd}(E')\), which finally yields \(H \in \text{outcont}(H^*,\mathcal{A}_s,k)\). \(\square\)

Proof of Lemma 15. Let \(\mathcal{A}_i \subseteq_{\text{TD}} \mathcal{A}_s\) then for every \(H \in \text{trd}(\mathcal{A}_s,k)\) we also have \(H \in \text{trd}(\mathcal{A}_i,k)\). So pick \(m \in \mathbb{N}\), let \(H^* \in \text{trd}(\mathcal{A}_s,m)\) and take \(H \in \text{outcont}(H^*,\mathcal{A}_s,m) \subseteq \text{trd}(\mathcal{A}_s,m+1)\). We want to show that \(H \in \text{outcont}(H^*,\mathcal{A}_s,m)\).
By assumption we know that $H \in \text{trd}(\mathcal{A}_s, m + 1)$. In particular that means there must be at least one adversary $E \in \text{adv}(\mathcal{A}_s, m + 1)$ such that $\text{trd}(E) = H$. However, for this adversary, we know that $H^* \sqsubseteq_m \text{trd}(E)$ and for all $\sigma \in L^m L_I$ we have $P_{\text{trd}(E)}(\sigma) = 0$ and by trace distribution inclusion $\text{trd}(E) = H$. Thus $H \in \text{outcont}(H^*, \mathcal{A}_s, m)$ and therefore $\mathcal{A}_L \sqsubseteq_{\text{pioco}} \mathcal{A}_S$. 

\textbf{Proof of Theorem 16.} \(\implies\) Let $\mathcal{A}_L \sqsubseteq_{\text{pioco}} \mathcal{A}_S$, fix $m \in \mathbb{N}$ and take a trace distribution $H^* \in \text{trd}(\mathcal{A}_L, m)$. To show that $H^* \in \text{trd}(\mathcal{A}_L, m)$, we prove that every prefix of $H^*$ is in $\text{trd}(\mathcal{A}_L, m)$, i.e. if $H^* \sqsubseteq_k H^*$ for some $k \in \mathbb{N}$, then $H^* \in \text{trd}(\mathcal{A}_L)$. The proof is by induction up to $m \in \mathbb{N}$ over the prefix trace distribution length, denoted by $k$.

Obviously $H^* \in \text{trd}(\mathcal{A}_L, 0)$ yields both $H^* \sqsubseteq_0 H^*$ and $H^* \in \text{trd}(\mathcal{A}_L)$. Now assume, we know that $H^* \sqsubseteq_k H^*$ for some $k < m$ and $H^* \in \text{trd}(\mathcal{A}_L)$. Furthermore let $H'' \in \text{trd}(\mathcal{A}_L, k + 1)$, such that $H'' \sqsubseteq_{k+1} H^*$. If we can show that $H'' \in \text{trd}(\mathcal{A}_L, k + 1)$, we are done.

With $H^* \in \text{trd}(\mathcal{A}_L, k)$, we take $H''' \in \text{outcont}(H^*, \mathcal{A}_L, k)$ such that all traces of length $k + 1$ ending in an output action have the same probability, i.e. for all $\sigma = \ell_{k}^2 \delta_{k}$, we have $P_{H'''}(\sigma) = P_{H''}(\sigma)$. By assumption $\mathcal{A}_L \sqsubseteq_{\text{pioco}} \mathcal{A}_S$ and thus $H''' \in \text{outcont}(H^*, \mathcal{A}_L, k) \sqsubseteq \text{trd}(\mathcal{A}_L)$.

Let $E \in \text{adv}(\mathcal{A}_L, k + 1)$ the corresponding adversary such that $\text{trd}(E) = H'''$. By construction, we have $P_{\text{trd}(E)}(\sigma a!_1) = P_{H'''}(\sigma a!)$ and $P_{\text{trd}(E)}(\sigma b?) = 0 \neq P_{H'''}(\sigma b?)$ for all $\sigma \in L^k$. We create yet another adversary, denoted by $E' \in \text{adv}(\mathcal{A}_L, k + 1)$ such that for all $\sigma \in L^k$ and $a!_1 \in L^2_\delta$, we have $P_{\text{trd}(E')}(\sigma) = P_{\text{trd}(E')}(\sigma a!) = P_{\text{trd}(E')}(\sigma a!)$. Taking the sum over all probabilities of those traces yields

$$\sum_{a!_1 \in L^2_\delta} P_{\text{trd}(E)}(\sigma a!) = 1 - \alpha,$$

where $\alpha \in [0, 1]$ and consequently the remaining bit is covered by

$$\sum_{b? \in L^2_{\delta}} P_{H'''}(\sigma b?) = \alpha.$$

The aim is now to set the behaviour of $E'$ such that $\sigma \in L^k L_I$ has $P_{H'''}(\sigma) = P_{\text{trd}(E')}(\sigma)$. We prove that this can indeed be done independently from $\sigma$. The input enabledness gives that for all $\sigma b? \in \text{traces}(\mathcal{A}_L)$, we also have $\sigma b? \in \text{traces}(\mathcal{A}_S)$. Assume $P_{H'''}(\sigma) = p$ and thus

$$\alpha = \sum_{b? \in L^2_{\delta}} P_{H'''}(\sigma b?) = P_{H'''}(\sigma b_1?) + \ldots + P_{H'''}(\sigma b_n?) = p\alpha_1 + \ldots + p\alpha_0$$

$$= P_{\text{trd}(E')}(\sigma b_1?) + \ldots + P_{\text{trd}(E')}(\sigma b_n?).$$

However, since $\text{trd}(E) \sqsubseteq_k H''$, we also have $P_{\text{trd}(E)}(\sigma) = p$.

The last detail not yet specified about $E'$ is the behaviour of paths of length $k + 1$ ending in an input transition. We demonstrate the choice of $E'$ for $p\alpha_1 = P_{\text{trd}(E')}(\sigma b_1?)$, and denote the associated paths $\{\pi_1, \ldots, \pi_n\} = \text{trace}^{-1}(\sigma)$. Furthermore $\pi'_j := \pi_j \mu b_1 s_{i_j}$ for some $s_{i_j} \in S$, $j = 1, \ldots, l$, which are reachable after $\pi_i$ and distributions containing $b?$. Thus we want
\begin{align*}
p\alpha_1 & = P_{\text{trd}(E')}(\sigma b?) = \sum_{i=1}^{n} P_{E'}(\pi'_i) \\
& = \sum_{i=1}^{n} \sum_{j=1}^{l} Q^{E'}(\pi_i) E'(\pi'_i)(\mu) \mu(b_1?,s_i)_{\alpha_1} \\
& = p\alpha_1 \sum_{i=1}^{n} \sum_{j=1}^{l} \mu(b_1?,s_i)_{\alpha_1}.
\end{align*}

We can do the same for all \(\alpha_i\) for \(i = 1,\ldots,\omega\). Note that the choice of the adversary does not depend on the chosen trace \(\sigma\) but solely on the presupposed behaviour of \(H''\). Thus we have found \(E' \in \text{adv}(\mathcal{A}_s,k+1)\) such that \(\text{trd}(E') = H''\). Hence \(H'' \in \text{trd}(\mathcal{A}_s,k+1)\), which ends the induction. Since this is possible for every \(m \in \mathbb{N}\), we get \(\mathcal{A}_s \subseteq \pioco \mathcal{A}_s\), ending the proof.

\[\Rightarrow\] See Lemma 15 for the proof. In particular we do not even require input enabledness for \(\mathcal{A}_s\) in this case.

**Proof of Theorem 17.** Let \(\mathcal{A} \subseteq \pioco \mathcal{B}\) and \(\mathcal{B} \subseteq \pioco \mathcal{C}\) and \(\mathcal{A}\) and \(\mathcal{B}\) be input enabled. By Theorem 16 we know, that \(\mathcal{A} \subseteq \text{TD} \mathcal{B}\). So let \(k \in \mathbb{N}\) and \(H^* \in \text{trd}(\mathcal{A},k)\). Consequently also \(H^* \in \text{trd}(\mathcal{B},k)\) and thus the following embedding holds

\[\text{outcont}(\mathcal{A},H^*,k) \subseteq \text{outcont}(\mathcal{B},H^*,k) \subseteq \text{outcont}(\mathcal{C},H^*,k),\]

and thus \(\mathcal{A} \subseteq \pioco \mathcal{C}\).
A Visual Formalism for Interacting Systems

Paul C. Jorgensen
School of Computing and Information Systems
Grand Valley State University
Allendale, Michigan USA
jorgensp@gvsu.edu

Interacting systems are increasingly common—many examples pervade our everyday lives: automobiles, aircraft, defense systems, telephone switching systems, financial systems, national governments, and so on. Closer to computer science, embedded systems and Systems of Systems are further examples of interacting systems. Common to all of these is that some “whole” is made up of constituent parts, and these parts interact with each other. By design, these interactions are intentional, but it is the unintended interactions that are problematic. The Systems of Systems literature uses the terms “constituent systems” and “constituents” to refer to systems that interact with each other. That practice is followed here.

This paper presents a visual formalism, Swim Lane Event-Driven Petri Nets, that is proposed as a basis for Model-Based Testing (MBT) of interacting systems. In the absence of available tools, this model can only support the offline form of Model-Based Testing.

1 Existing Models for Interacting Systems

To support offline MBT of interacting systems, a model must be capable of expressing the ways in which constituents interact. The best known, and most widely used, such model is the Statechart model [3]. Statecharts have been incorporated into the Unified Modeling Language (UML) and further codified into Types, I, II, and III of UML Statecharts. The model presented in this paper has been shown to be formally equivalent [2] to all three types of UML statecharts.

Statecharts contain orthogonal regions, and these nicely represent distinct devices. As such, they can also be used to represent systems that interact with each other. The statechart broadcasting mechanism is the only vehicle for communicating interactions among the orthogonal devices/components. There is an elaborate language on transitions among blobs in an orthogonal region. Taken together, these notations result in a very “dense” model that is best understood by executing the statechart with an engine. There can be no doubt about the expressive power of statecharts. In a conversation [5] at a Grand Rapids (Michigan) avionics company, representatives of the i-Logix company related a success story in which exhaustive execution of a statechart model of a fully-deployed ballistic missile launch control system revealed a legitimate sequence of events that would launch a missile that was not known to the developing defense contractor.

Contemporaneously with Harel’s work on statecharts, a North American industry group proposed the Extended Systems Modeling Language (ESML) which described how activities in a traditional data flow diagram could communicate with each other [1]. The ESML prompts allowed an activity in a data flow diagram to Enable, Disable, Activate, Pause, Resume, Suspend, or Trigger another activity. Synonyms of many of these verbs are available in the statechart transition language.
2 Comparing Statecharts with Swim Lane Event-Driven Petri Nets

While they are clearly a powerful modeling technique, there are some problems with statecharts (as originally defined). Statecharts:

- Are best understood when executed by a customer using a statechart engine,
- Are a top-down model,
- Can only be composed under very limited circumstances [4], and
- Are both rigorous and complex, a five-day training course is recommended [6].

These limitations are all answered by Swim Lane Event-Driven Petri Nets, specifically they are:

- intuitively clear, once the basic mechanism of Petri Net transition firing is understood,
- a bottom-up model. As such, they work well in agile developments.
- easily composed, particularly if the composition is accomplished in a database with well-designed queries.
- easy to learn

3 Event-Driven Petri Nets and Swim Lane Petri Nets

Basic Petri nets need two slight enhancements to become Event-Driven Petri Nets (EDPNs) [7]. The first enables them to express more closely event-driven systems, and the second deals with Petri net markings that express event quiescence, an important notion in object-oriented applications. Taken together, these extensions result in an effective, operational view of software requirements.

**Definition:** An Event-Driven Petri Net (EDPN) is a tripartite-directed graph \((P, D, S, In, Out)\) composed of three sets of nodes, \(P\), \(D\), and \(S\), and two mappings, \(In\) and \(Out\), where:

- \(P\) is a set of port events
- \(D\) is a set of data places
- \(S\) is a set of transitions
- \(In\) is a set of ordered pairs from \((P \cup D) \times S\)
- \(Out\) is a set of ordered pairs from \(S \times (P \cup D)\)

The drawing conventions for EDPNs are in Figure 1. Other than explicit representation of discrete events, EDPNs are very similar to ordinary Petri nets. The other main difference is that an ordinary Petri net is a closed system, in which tokens and markings are determined only by transition firing. EDPNs support a concept of event quiescence, which is an extension to deadlock in an ordinary Petri net. Since events are (usually) from external devices, an EDPN more accurately represents an event-driven system (so EDPNs represent "open" systems.) Both ordinary and Event-Driven Petri Nets can be placed into UML-style "swim lanes." The most convenient interpretations of swim lanes is that they "contain" interacting constituent systems.
4 Communication Primitives for Interacting Systems

Integration testing for a single system is based on the assumption that the units being integrated have all been individually, and thoroughly, tested. The usual goals of integration testing are to find faults that are due to interfaces among the units, and as such, would not be revealed by unit testing. This clearly extends to testing interacting systems where we assume that the constituent systems are all thoroughly tested and function correctly. The goal of testing for interacting systems is to focus on the ways in which constituent systems communicate; here we present a set of communication primitives for that purpose. The first two are from ordinary Petri nets, followed by the communication primitives from the Extended Systems Modeling Language (ESML) [1]. The ESML primitives are supplemented by three primitives that represent service requests among interacting systems. Each primitive is briefly described and illustrated by accompanying figures next.

4.1 Petri Net Conflict

Figure 2 shows the conflict pattern of ordinary Petri nets. With the given marking, both transitions are enabled. Firing either one disables the other. In EDPNs, if place p2 is replaced by a port input event, we have a context-sensitive input event, where outputs of the same physical input depend on the context in which the event occurs.

4.2 Petri Net Interlock

The Petri net interlock pattern is used to set a priority. With the marking shown in Figure 3, the secondary action cannot fire until the preferred action fires. If these are linked, we have the pattern for mutual exclusion.

4.3 ESML Enable, Disable, and Activate

The work of the ESML committee began with activities in a data flow diagram that could control other activities. Here the place labeled "e/d" is used to enable, and later, to disable the controlled action in Figure 4. Clearly the enabling and disabling actions are from other constituents. Once a transition is enabled, it remains enabled due to the output leading back to the e/d place. Notice that the enable/disable
place establishes Petri net conflict between the controlled and disable actions. The enable portion essentially gives permission for the controlled action to occur, but in terms of marking, it may have to await additional inputs. The edge from the controlled action back to the enable/disable place assures that an enabled transition remains enabled. The activate prompt is simply a sequence of enable followed by a disable.

### 4.4 ESML Trigger

The ESML trigger prompt is stronger than the enable prompt. It essentially requires the controlled action to fire as soon as it is fully enabled, as in Figure 5. A trigger prompt can be paired with a disable prompt. As with the Enable prompt, once a controlled action has been started by a Trigger Prompt, the Trigger place (marked t in Figure 5) remains marked due to the output that leads back to it. (Of course, this assumes that the other inputs are still available to the controlled action. As with the Enable prompt, a Trigger prompt can be removed by a Disable prompt.

### 4.5 ESML Suspend, Resume, and Pause

The ESML suspend, resume, and pause prompts (Figure 5) were designed to interact with an ongoing activity without losing any of the work done prior to the point of suspension. The suspend place “s” is actually an interlock to assure that the resume action must follow a suspend action. The suspend action can be used to transfer temporary control to a more important action. As with activate, the pause prompt is a sequence of suspend followed by resume.

### 4.6 Service Requests

The last three patterns are directed at communication among constituents. When constituent A requests a service from constituent B, the request place is similar to an enable prompt (Figure 6). Another interpretation is that a request is a message. Once a request is made, constituent A awaits a response from constituent B.
4.6.1 Accept and Reject Requests

On receipt of a request, constituent B may choose to accept or reject the request. When the service has been provided, constituent B returns a "done" response to the waiting constituent A (see Figure 8). Similarly, if constituent B rejects the request, constituent B returns a "not done" response to the waiting constituent. Notice that the request place is in Petri net conflict with respect to the two responses.

Figure 7: SoS Accept and Reject responses to a Request

4.6.2 Postpone Request

Consider situations in which a constituent may have the latitude to postpone a request. This most likely happens because the constituent has more urgent tasks. Note the use of an interlock to show the task priority in Figure 8.
5 Example: a Garage Door Controller

A system to control a motorized garage door is comprised of several components: a drive motor, the garage door wheel tracks, and a wireless control device. There are two safety features, a light beam near the floor, and an obstacle sensor. These latter two devices operate only when the garage door is closing. If the light beam is interrupted (possibly by a pet) the door immediately stops, and then begins to open. Similarly, if the door encounters an obstacle while it is closing (say a child’s bicycle left in the path of the door), the door stops and reverses direction as with a light beam interruption. There is a third way to stop a door in motion, either when it is closing or opening a signal from the wireless control device. In response to this signal the door stops in place. A subsequent signal starts the door in the same direction as when it was stopped. Finally, there are sensors that detect when the door has moved to one of the extreme positions, either fully open or fully closed. Figure 9 is a SysML context diagram of the garage door controller.

This example is deliberately small, yet it suffices to illustrate several of the previously described interactions. Here are a few examples:
• When the door is open, a signal from the wireless keypad triggers the drive motor.
• When the door is in motion, either opening or closing, a signal from the wireless keypad triggers the motor to stop. (Note: this portion of the problem could be interpreted as an ESML Pause.)
• When the door is closing, the Light Beam and the Obstacle Sensors are enabled.
• The fully closed sensor disables the Light Beam and the Obstacle Sensors.
• An input from either the Light Beam or the Obstacle Sensor triggers the drive motor to stop and then begin opening.

5.1 Statechart model of the Garage Door Controller

A full statechart model is given in Figure 10. The orthogonal regions describe local views of the full garage door, the motor, and both safety sensors. The input events and output actions are:
e1: wireless control signal
  a1: start drive motor down
e2: light beam interruption sensed
  a2: start drive motor up
e3: obstacle sensed
  a3: stop drive motor
e4: end of down track reached
  e5: end of up track reached

Some of the transitions are marked with output actions lettered a, b, c, d, and e. These will illustrate the broadcasting mechanism of statecharts. The sequence corresponds to the Statechart execution for the following scenario: pre-condition: door is up, Input event sequence: e1, e1, e1, e2, e5, and the post-condition: door is up. This corresponds to what an offline model-based tester would do to identify a test case for the scenario. Good practice dictates making an “execution table,” as shown in Table 1.

<table>
<thead>
<tr>
<th>Step</th>
<th>Door</th>
<th>Motor</th>
<th>Light Beam Sensor</th>
<th>Obstacle Sensor</th>
<th>Input Event</th>
<th>Broadcast Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>s1</td>
<td>s9</td>
<td>s11</td>
<td>s15</td>
<td>e1</td>
<td>a</td>
</tr>
<tr>
<td>1</td>
<td>s5</td>
<td>s10</td>
<td>s12, s13</td>
<td>s16, s17</td>
<td>e1</td>
<td>b</td>
</tr>
<tr>
<td>2</td>
<td>s3</td>
<td>s9</td>
<td>s11</td>
<td>s15</td>
<td>e1</td>
<td>c</td>
</tr>
<tr>
<td>3</td>
<td>s5</td>
<td>s10</td>
<td>s12, s13</td>
<td>s16, s17</td>
<td>e2</td>
<td>d</td>
</tr>
<tr>
<td>4</td>
<td>s6</td>
<td>s8</td>
<td>s12, s14</td>
<td>s15</td>
<td>e5</td>
<td>e</td>
</tr>
<tr>
<td>5</td>
<td>s1</td>
<td>s9</td>
<td>s12, s14</td>
<td>s15</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

To use a statechart for offline MBT, the tester must begin with a full statechart that shows only the input events that cause transitions in the orthogonal regions. Next, a scenario is postulated, and an exercise similar to the one above is followed to create an execution table. This is then the skeleton of a MBT test case. If the tester has access to a statechart engine, the process is greatly simplified; the tester simply selects a starting condition and defines a sequence of inputs. The engine produces an execution table similar to Table 1.
5.2 Swim Lane EDPN models of the Garage Door Controller

Table 2 contains a legend for all the EDPN elements in Figures 11, 12, 13, 14, and 16. The basic operation of the Garage Door Controller is shown as an Event-Driven Petri Net in Figure 12 (the intermediate stopping and the safety devices are omitted). Places d1: Door Up and d4: Door Down are contexts for input event p1: wireless keypad signal. Since d1 and d4 are mutually exclusive, the context sensitivity is resolved. For now, assume the door is fully open (d1 is marked). When the transition t1 fires, output event p7: start drive motor down occurs, and the door is in the state d2: Door Closing. After some time interval (13 seconds in my garage) the door reaches the end of the down track, which is represented here as the input event p2. When transition t2 fires, the output p9: stop drive motor occurs, and the Garage Door Controller is in the fully closed state d4. If event p1 occurs again, transition t3 can fire, which causes output event p8 to occur and leaves the garage door in the state d5: Door Opening. Once the end of the up track is reached, input event p3 occurs, transition t4 fires, the motor is stopped (output event p9),
Table 2: Swim Lane EDPN elements for the Garage Door Controller

<table>
<thead>
<tr>
<th>Input events</th>
<th>Output events (actions)</th>
<th>Data Places</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1: wireless keypad signal</td>
<td>p7: start drive motor down</td>
<td>d1: Door Up</td>
</tr>
<tr>
<td>p2: end of down track hit</td>
<td>p8: start drive motor up</td>
<td>d2: Door Closing</td>
</tr>
<tr>
<td>p3: end of up track hit</td>
<td>p9: stop drive motor</td>
<td>d3: Door Stopped going down</td>
</tr>
<tr>
<td>p4: brief motor pause</td>
<td></td>
<td>d4: Door Down</td>
</tr>
<tr>
<td>p5: light beam sensor</td>
<td></td>
<td>d5: Door Opening</td>
</tr>
<tr>
<td>p6: obstacle sensor</td>
<td></td>
<td>d6: Door Stopped going up</td>
</tr>
</tbody>
</table>

Table 3: Selected Paths in Figure 10

<table>
<thead>
<tr>
<th>Path Description</th>
<th>Transition Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Close an open garage door.</td>
<td>t1, t2</td>
</tr>
<tr>
<td>2. Open a closed garage door.</td>
<td>t3, t4</td>
</tr>
<tr>
<td>3. Open closed door and then close it.</td>
<td>t3, t4, t1, t2</td>
</tr>
</tbody>
</table>

and the door is back in the fully open state (d1). When described as an Event-Driven Petri Net (without swim lanes), we have the full picture, but we miss the interactions among devices. The interactions are apparent, and we could show the overall execution by a marking sequence.

System test cases can be derived directly from an Event-Driven Petri Net. A system test case corresponds to a sequence of transitions that fire. Since the EDPN in Figure 11 is 3-connected (a true path exists to and from every transition) there can be a countable infinite set of distinct paths. Table 3 lists three sample paths.

Deriving a full system test case is straightforward. The test case for Path 1 in Table 2 is:

**Name:** Close an open garage door.

**Pre-conditions:** Garage Door is open

**Event Sequence**

<table>
<thead>
<tr>
<th>Input Events</th>
<th>Output Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. p1: wireless keypad signal</td>
<td>2. p7: start drive motor down</td>
</tr>
<tr>
<td>3. p2: end of down track hit</td>
<td>4. p9: stop drive motor</td>
</tr>
</tbody>
</table>

**Post-conditions:** Garage Door is closed

Figure 12 shows a detailed view of the interaction between the garage door closing and the enabling/disabling of the two safety features. This view is more appropriate for offline model-based testing, as it allows the tester to focus on specific interactions. In Figure 13, the interactions between the motor and the two safety devices are shown. This can be considered to be a continuation of the enabling and disabling of the safety features shown in Figure 13. If either safety device input occurs, the Trigger prompt immediately causes the motor to reverse and drive the garage door to the open position. Notice that Figures 13 and 14 could be composed into a larger, more expressive, Swim Lane EDPN. This becomes cumbersome, and even unwieldy quickly, as we see in Figure 14.

For comparison, look at Figure 14 – a fairly complete Swim Lane Event-Driven Petri Net showing the enabling of the safety features (light beam and obstacle sensor) and how an event from either device stops and then reverses the garage door motion. In Figure 14, assume an initial marking of place d1:
Door Up. If event p1: wireless keypad signal occurs, transition t1 fires with four results: the light beam sensor and the obstacle sensors are enabled, the drive motor is started in the down (closing) direction, and the Garage Door is in the d2: Door Closing state. If event p5: light beam sensor occurs, transition t3 fires, causing output event p9: stop drive motor, leaving the garage door in state d3: Door Stopped going down. (The scenario for the obstacle sensor is symmetric to that for the light beam sensor.) The next event is p4: brief motor pause, which allows transition t4 to fire because it has been triggered, and this causes output event p8: start drive motor up to occur, leaving the garage door in state d5: Door Opening. Event p3 occurs when the end of the up track is reached, which stops the drive motor (p9) leaving the garage door in state d1: Door Up.

6 Concluding Thoughts

Swim Lane Event Driven Petri Nets provide several advantages for model-based testing:

- they are a bottom-up approach
- they can be easily composed
- they permit focused description of interactions among constituents
- they can be used for automatic derivation of system test cases
- they support a useful hierarchy of system test coverage metrics.
The biggest limitation of Swim Lane Event Driven Petri Nets is that the drawings do not scale up well. They are vulnerable to a diagrammatic explosion similar to the "finite state machine explosion." All (graphical) Petri net models suffer from the problem of space-consuming diagrams. Figure 15 presents an elegant answer to this issue. Rather than compose Swim Lane EDPN diagrams, we can populate a database with the E/R description in Figure 15. With such a formulation, questions of connectivity are reduced to well-constructed database queries. One clear advantage of this is that now there is no practical limit to Swim Lane EDPN composition.

6.1 Graphical Composition of Event-Driven Petri Nets

Graphical composition of Event-Driven Petri Nets is straightforward, but it easily expands to spatial difficulty. Figure 16 shows two Event-Driven Petri Nets for closing and opening the garage door; their composition was shown earlier in Figure 11.
6.2 Composition of Event-Driven Petri Nets in the E/R Model Database

<table>
<thead>
<tr>
<th>Database for Door Closing EDPN</th>
<th>EventInput</th>
<th>EventOutput</th>
<th>DataInput</th>
<th>DataOutput</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event</td>
<td>Transition</td>
<td>Event</td>
<td>Transition</td>
<td>Data</td>
</tr>
<tr>
<td>p1</td>
<td>t1</td>
<td>P7</td>
<td>t1</td>
<td>d1</td>
</tr>
<tr>
<td>p2</td>
<td>t2</td>
<td>P9</td>
<td>t2</td>
<td>d2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Database for Door Opening EDPN</th>
<th>EventInput</th>
<th>EventOutput</th>
<th>DataInput</th>
<th>DataOutput</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event</td>
<td>Transition</td>
<td>Event</td>
<td>Transition</td>
<td>Data</td>
</tr>
<tr>
<td>p1</td>
<td>t3</td>
<td>p7</td>
<td>t3</td>
<td>d3</td>
</tr>
<tr>
<td>p3</td>
<td>t4</td>
<td>p8</td>
<td>t4</td>
<td>d4</td>
</tr>
</tbody>
</table>
Figure 14: Swim Lanes for Light Beam and Obstacle Sensor Enabling

<table>
<thead>
<tr>
<th>EventInput</th>
<th>EventOutput</th>
<th>DataInput</th>
<th>DataOutput</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event</td>
<td>Transition</td>
<td>Event</td>
<td>Transition</td>
</tr>
<tr>
<td>p1</td>
<td>t1</td>
<td>p6</td>
<td>t1</td>
</tr>
<tr>
<td>p1</td>
<td>t3</td>
<td>p7</td>
<td>t3</td>
</tr>
<tr>
<td>p2</td>
<td>t2</td>
<td>p8</td>
<td>t2</td>
</tr>
<tr>
<td>p3</td>
<td>t4</td>
<td>p8</td>
<td>t4</td>
</tr>
</tbody>
</table>
6.3 Further Guidelines

As a guideline, Swim Lane EDPNs are best used to focus on particular interactions, and then using the database approach to keep the overall model. While test cases can be derived by inspection of a graphical model, this is more difficult from the underlying database. Finally, Swim Lane EDPNs support the definition of the set of system level test coverage metrics given in Table 4. These metrics all refer to a set of test cases \( T \) derived from a corresponding Swim Lane EDPN description.
Table 4: Test Coverage Metrics

<table>
<thead>
<tr>
<th>Test Cover</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ct</td>
<td>every transition</td>
</tr>
<tr>
<td>Cp</td>
<td>every data place</td>
</tr>
<tr>
<td>Cie</td>
<td>every input event</td>
</tr>
<tr>
<td>Coe</td>
<td>every output event</td>
</tr>
<tr>
<td>Ccontext</td>
<td>Cie in every context</td>
</tr>
</tbody>
</table>

References


Potential Errors and Test Assessment in Software Product Line Engineering

Hartmut Lackner  
Humboldt-Universität zu Berlin  
Germany  
lackner@informatik.hu-berlin.de

Martin Schmidt  
Humboldt-Universität zu Berlin  
Germany  
schmidma@informatik.hu-berlin.de

Software product lines (SPL) are a method for the development of variant-rich software systems. Compared to non-variable systems, testing SPLs is extensive due to an increasingly amount of possible products. Different approaches exist for testing SPLs, but there is less research for assessing the quality of these tests by means of error detection capability. Such test assessment is based on error injection into correct version of the system under test. However to our knowledge, potential errors in SPL engineering have never been systematically identified before.

This article presents an overview over existing paradigms for specifying software product lines and the errors that can occur during the respective specification processes. For assessment of test quality, we leverage mutation testing techniques to SPL engineering and implement the identified errors as mutation operators. This allows us to run existing tests against defective products for the purpose of test assessment. From the results, we draw conclusions about the error-proneness of the surveyed SPL design paradigms and how quality of SPL tests can be improved.

1 Introduction

Software product line (SPL) engineering is an emerging method for the development of variant-rich software systems. Based on a SPL specification single products can be configured and derived. SPL engineering is a systematic and planned process to reuse software artifacts most efficiently [8]. This also includes quality assurance, where one of the most important ones is testing. But testing a SPL is different from testing non-variable systems and thus is investigated intensively [27, 11, 28, 21]. Challenges in testing SPLs are the selection of products for testing and the design of tests from the SPL’s specification.

Though there are many methods proposed for testing a product line, until now, quality assessment of tests was limited to mutating individual products of the SPL. This approach has two major drawbacks: first, developers can introduce errors on all kinds of artifacts, not only on final products specifications. For better understanding we analyze different design paradigms for the specification of products from a SPL and the errors that can occur during the respective design processes. From the results, we developed mutation operators for variability models, and domain models that mimic possible faults in these models.

Secondly, the selection of products and subsequently its mutations is biased by the available tests, since only products for which tests are available will be tested. Therefore, mutation analysis assesses the quality of the tests for particular products, but not for the whole SPL. In contrast, we define a mutation system and operators on the domain engineering-level. This enables us to assess the test quality independently from the tested products. Subsequently, the test quality over the complete SPL is assessed.

The remainder of this article is structured as follows: In Section 2 we summarize the foundations of SPL engineering and test assessment. In Sec. 3 we define and classify kinds of errors. We present our SPL test assessment system and the evaluation of three examples in Sec. 4. Eventually, we show related work in Sec. 5 and conclude in Sec. 6.
2 Preliminaries

In this section, we present the foundations that our work is based on. First, we give a short introduction for model-based product line engineering. The second part is about mutation analysis. The third part deals with potential errors and mutation in software development.

2.1 Model-based Product Line Engineering

Individual customer expectations and the reuse of existing assets in a product’s design are two driving factors for the emergence of product line engineering: increasing the number of product features while keeping system engineering costs at a reasonable level. In terms of software engineering, a SPL is a set of related software products that share a common core of software assets (commonalities), but can be distinguished (variabilities) [29].

The definition and realization of commonalities and variabilities is the process of domain engineering. Actual products are built during application engineering (cf. fig. 1). Here, products are built by reusing domain artifacts and exploiting the product line variability.

Like many methodologies, SPL engineering can be supported by model-based abstractions such as feature models. Feature models offer a way to overcome the aforementioned challenges by facilitating the explicit design of global system variation points [18]. In consequence, variation points are not spread across one or multiple domain models anymore, but instead linked to one core of variability description.

A feature model has a tree structure in which a feature can be decomposed into sub-features. Fig. 2 shows an example feature model, that will also be used as an example in section 4. A parent feature can have the following relations to its sub-features: (a) Mandatory: child feature is required, (b) Optional: child feature is optional, (c) Or: at least one of the children features must be selected, and (d) Alternative: exactly one of the children features must be selected. Furthermore, one may specify additional (cross-tree) constraints between two features A and B: (i) A requires B: the selection of A implies the selection of B, and (ii) A excludes B: both features A and B must not be selected for the same product.

A feature model captures the system’s variation points in a concise form. Its elements, however, are only symbols [9]. Their semantics has to be provided by mapping them to models with semantics. Such a mapping can be defined using an explicit mapping model. A mapping model consists of relations from feature model elements to domain model elements. We refer to the tuple of feature model, mapping model, and domain model as SPL specification.

Product models or code can be materialized from the SPL specification by providing a configuration. A configuration assigns a valuation to every feature in the feature model, denoting the presence or

---

Figure 1: Process of SPL engineering
Figure 2: A feature model for the eShop example.
absence of the mapped elements. The valuation must not violate the constraints imposed by the feature model.

Based on this setup three paradigms have established for specifying SPLs. These paradigms will be briefly introduced as follows.

2.1.1 Negative Variability

In this case, the domain model is designed in terms of a so called 150% model. A 150% model contains every element that is used in at least one product configuration and, thus, subsumes every possible product [14] (Fig. 3a). We consider the combination of a feature model, a mapping model, and a UML model as SPL specification.

Each mapping in the mapping model maps a single feature to a set of transitions. Additionally, each mapping has a Boolean flag that indicates whether the mapped model elements are part of the product when the feature is selected (true) or unselected (false). Figure 4 shows an excerpt of the eShop specification, where parts of the feature model are depicted in the upper half and parts of the state machines payment process are shown in the lower half. In between, we find a mapping, denoted by a dotted edge, from feature "Credit Card" to the transition labeled as "SelectCreditCard[1]/".

2.1.2 Positive Variability

In contrast to negative variability to design the domain model, positive variability starts with a minimal core that contains features that are common to all possible products. From this starting point additional features will be added by a designer (Fig. 3b).

2.1.3 Delta Modeling

Designing products in SPL engineering using positive or negative variability is called feature-oriented. In contrast to these paradigms, there is another approach which is referred to as delta modeling (also delta-oriented programming) [30]. Using delta modeling for the purpose of designing SPLs, two parts are needed. The first is a core module, that comprises a set of features that represent a valid product. The second part is a set of delta modules which specify changes that will be applied to the core module. These changes can either be the construction (add) or destruction (remove) of features (Fig. 3c).
2.2 Mutation Analysis

Mutation analysis (also mutation testing) as introduced by DeMillo et al. [10] is a error-based testing technique with the intended purpose to assess the quality of tests that will be applied to a system.

The process of mutation analysis seeds errors into software by creating modified versions of the original software, where each created version contains one error. After that existing test cases are used to execute the defective versions (mutants) with the goal to distinguish the defective ones (to kill a mutant) from the original software. The ratio of killed mutants to generated mutants is called mutation score, that will be computed after the execution of all test cases. The main goal of the test designer is to achieve the highest possible mutation score [26, 17].

Though mutation operators are applied to introduce errors, there is the chance, that the resulting mutant offers the same behavior like the original. This type of mutants are referred as hidden mutants. Although the detection of hidden mutants is an undecidable problem, hidden mutants are supposed to be removed from the mutation analysis before scoring is performed [17].

According to Jian and Harman [16] we can distinguish multiple kinds of mutants that can be created. The simplest ones and already mentioned are first-order mutants that have only one introduced error. Even if first-order mutants can be killed during the process of mutation testing, this does not guarantee that a combination of two (or even more) mutants will also be detected by the test suite. Such combined mutants are referred as higher-order mutants.

2.3 Potential Errors and Mutations

In mutation analysis, defective software versions are derived from a set of potential errors a human can make during software development. Potential errors are implemented as mutation operators, which are applied to the original software for introducing errors. The mutation operator’s design affects the validity of the resulting mutation scores and the costs for testing by means of the amount of mutants to create and the number of tests to execute against them. Thus, we apply the following four guiding principles for creating mutation operators [5, 34]:

1. Mutation categories should model potential error. It is important to recognize different types of error. In fact, each mutation operator is designed to model errors belonging to the corresponding error class.
2. Only simple, first-order mutants should be generated. These mutants are produced by making exactly one syntactic change to the original specification. This restriction is justified by the coupling effect hypothesis which says that the test sets that detect simple mutants will also detect more complex mutants [23].
3. Only syntactically and semantically correct mutants should be generated. Some mutations may result in an illegal expression, such as division by 0. Such mutants should not be generated.
4. Do not produce too many mutants. This includes some practical restrictions. For example, do not replace a relational connector with its opposite, if for other mutants a term negation operator is applied, since both mutants are semantically equivalent.

From other mutation systems [4, 1, 12], we identified the following general categories for model-based mutation operators:

1. Model element deletion: a model designer forgets to add a model element, e.g. a feature, a mapping, or a transition
2. Model element insertion: a model designer inserts a superfluous model element, e.g. a feature, a mapping, or a transition

3. Property change: a model designer chooses a wrong value for a property of a model element, e.g. mandatory feature instead of optional, inverse value for a feature’s status, or wrong transition target.

For each model element-type, like mappings, transitions, guards, etc., one can check for applicable categories and implement mutation operators accordingly.

3 Potential Errors in Model-Based Product Line Engineering

In this contribution, we focus on errors in the feature mapping. The feature mapping has a major impact on the outcome of the product line’s materializations, however the design is complex and error-prone. We identify potential errors in a systematic way by checking each modeling paradigm for possibilities to add superfluous or omit necessary elements or change the value of an element’s attribute. For each potential error we discuss its effects onto the materializations.

Furthermore, according to the consequences of each errors for affected products, we assign one of the following four types to each potential error:

- **add** extends the behavior of affected products.
- **omit** restricts the behavior of affected products.
- **alter** extends and restricts the behavior of some products.
- **mix** extends, restricts, or both the behavior of affected products, depending on the model’s contents.

**Negative Variability**

In the negative variability paradigm, we identify the following model elements for potential errors from the feature mapping model: mappings, their attribute feature value, mapped feature, and the set of mapped elements. The errors which can be made on these model elements and their effects are as follows:

N1) **Omitted mapping**: a necessary mapping is left out by its entirety. Mapped elements will be part of every product unless they are restricted by other features. As a result, some or all products unrelated to the particular feature will include superfluous behavior. Products including the mapped feature are not affected, since the behavior was enabled anyway.

N2) **Superfluous mapping**: a superfluous mapping is added, such that a previously unmapped feature is now mapped to some domain model elements. This may also include adding a mapping for an already mapped feature, but with inverted feature value. Adding a mapping with feature value set to true results in the removal of elements from products unrelated to the mapped feature. Contrary, adding a mapping with feature value set to false removes elements from any product which the mapped feature is part of. In any case the behavior of at least some products is reduced.

N3) **Omitting a mapped element**: a mapped model element is missing from the set of mapped element in a mapping. Subsequently, a previously mapped element will not only be available in products which the said feature is part of, but also in products unrelated to this feature. As a result, some products offer more behavior than they should or contain unreachable model elements.
N4) *Superfluously mapped element:* an element is mapped although it should not be related to the feature it is currently mapped to. As a result the element becomes unavailable in products which do not include the associated feature. The product’s behavior is hence reduced.

N5) *Swapped feature:* the associated features of two mappings are mutually exchanged. Subsequently, behavior is exchanged among the two features and thus, affected products offer different behavior than expected. The result is the same as exchanging all mapped elements among two mappings.

N6) *Inverted feature status:* the bit-value of the feature value attribute is flipped. The mapped elements of the affected mapping become available to products where they should not be available. At the same time, the elements become unavailable in products where they should be. For example, if the feature value is true and is switched to false, the elements become unavailable to products with the associated feature and available to any product not including the said feature. Of course, other feature mappings to the same element(s) must still be considered.

**Positive Variability**

In SPL modeling with positive variability, a mapping is a bijection between features and modules composed from domain elements. Potential errors in the feature mapping models can be made at: mappings, mapped feature, and mapped module. We identify the following potential errors:

P1) *Omitted mapping:* a necessary mapping is missing in its entirety. This appears to us to be an unrealistic scenario, since one can automatically check for all modules being mapped to some feature. But if we consider the case of a missing mapping, products with the associated feature would be missing the modules functionality.

P2) *Superfluous mapping:* a superfluous mapping is added. Similar to the above, this is an unrealistic scenario for same reason: all modules should be mapped exactly once. In a model-based environment, this check should be easily automatable. However, if adding a superfluous mapping is possible, more behavior becomes enabled in products containing the mapping’s feature.

P3) *Swapped modules:* the associated modules of two mappings are mutually exchanged. As a result, all products containing one of the two features, but not the other, offer not the expected behavior. In contrast, all products containing none or both features behave as expected.

P4) *Swapped features:* the associated features of two mappings are mutually exchanged. The result is the same as above for swapped modules.

**Delta Modeling**

For other paradigms, like delta-modeling [30], we make similar observations. In contrast to positive variability models, delta-oriented variability models start from an actual core product, instead of a base module. From this on, only the differences from one product to another are defined by *deltas*. In delta-modeling, mapping multiple features to the same delta is allowed. A delta may add elements to and remove elements from the core product at the same time. As potential points of errors in delta-modeling we identify deltas, a delta’s set of mapped features, its set of removed elements from the base product, and its set of added elements.

D1) *Omitted delta:* the product line model misses an entire delta definition. Products containing features of the missing delta may lack behavior or offer to much of it. This depends on whether the delta removes and/or adds elements to the base product.
4 Product Line Test Assessment

As laid out in section 3, other errors can be made in model-based SPL engineering than in contrast to non-variable systems engineering. Furthermore, current test design methods and coverage criteria are not prepared to deal with these errors. To show the validity of our argument, we propose a mutation system for SPLs. It is specifically designed to assess test quality, by means of error detection capability (EDC), for the whole product line rather than for single systems. But mutation systems for SPLs need novel mutation operators. The reason for this is the separation of concerns in SPL engineering, where variability and domain engineering are split into different phases and models.

Mutation operators defined for non-variant systems cannot infer mutants including modules from other products, since this information is only available during domain engineering. However, we expect a high-quality test suite to detect such errors. Hence, we also propose new mutation operators based on the potential errors, we identified in section 3. For conciseness, we only consider potential errors from negative variability modeling for implementation.

4.1 Mutation System for SPLs

Performing mutation analysis on SPL tests is different from non-variant system tests, since in contrast to conventional mutation systems, a mutated SPL specification is not executable per se. Thus, testing
cannot be performed until a decision is made towards a set of products for testing. This decision depends on the SPL test suite itself, since each test is applicable to just a subset of products.

In Figure 5, we depict a mutation process for assessing SPL test suites, which addresses this issue. Independently from each other, we gain (a) a set of SPL specification mutants by applying mutation operators to the SPL specification and identify (b) a set of configurations describing the applicable products for testing. We apply every configuration in (b) to every mutant in (a), which returns a new set of product specification mutants. Any mutant structurally equivalent to the original product specification is immediately removed and does not participate in the scoring. The specification mutants are easily materialized into product mutants and finally, tests are executed. Our mutation scores are based on the SPL specification mutants, hence we established bidirectional traceability from any SPL specification mutant to all its associated product mutants and back again. If a product mutant is killed by a test, we backtrack its original SPL specification mutant and flag it as killed. The final mutation score is then calculated from the killed and the overall number of SPL specification mutants.

4.2 SPL Mutation Operators

Here, we present mutation operators for feature mapping models with negative variability. Furthermore, we enrich the mutation system by standard state machine operators and apply them on domain-level as well. For each operator, we describe how it was identified and its notion. Also, we discuss potentially invalid and hidden mutants resulting from each operator.

4.2.1 Feature Mapping

We design the mutation operators according to the potential errors identified in section 3. We do not consider inserting superfluous mappings as in this case it remains unclear which and how many UML elements should be selected for the mapping. We assume that this, if not carefully crafted, will lead to mostly invalid mutants.

*Delete Mapping (DMP)* The deletion of a mapping will permanently enable the mapped elements, if they are not associated to other features that constrain their enabledness otherwise. In our examples, no invalid mutants were created. However, for product lines that make heavy use of mutual exclusion (Xor and excludes) this does not apply. The reason for this are competing UML elements like transitions that would otherwise never be part of the same product. Multiple enabled and otherwise excluding transitions are possibly introducing non-determinism or at least unexpected behavior.

Some products mutants created with this operator might behave equivalent to an original product. This is the case for all products that include the feature for which the mapping was deleted.
**Delete Mapped Element (DME)** This operator deletes a mapped UML element from a mapping in the feature mapping model. It resembles the case, where a modeler forgot to map a UML element that should have been mapped.

Similar to the delete mapping operator, this operator may yield non-deterministic models, where otherwise excluding transitions are concurrently enabled. Product mutants equivalent to the original product model can be derived, if the feature associated to the deleted UML reference is part of the product.

**Insert Mapped Element (IME)** This operator inserts a new UML element to the mapping. This is the contrary case to the operators defined before, where mappings and UML elements were removed. However, inserting additional elements is more difficult than deleting them, since a heuristic must be provided for creating such an additional element. We decided to copy the first UML element reference from the subsequent mapping. If there are no more mappings, we take the first mapping. This operator is not applicable if there is just one mapping in the feature mapping model.

Again, there is a chance of creating invalid mutants: If a UML element reference is copied from a mutually excluded mapping, the resulting model may be invalid due to non-determinism.

**Swap Feature (SWP)** Swapping features exchanges the mapped behavior among each other. This operator substitutes a mapping’s feature by the following mapping’s feature and vice versa. The last feature to swap is exchanged with the very first of the model.

Non-deterministic behavior and thus invalid models may be designed by this operator. This is due to the fact that the mutation operator may exchange a feature from a group of mutually exclusive features by an unrestricted feature. In consequence, the previously restricted feature is now independent, while the unrestricted feature joins the mutually exclusive group. This may concurrently enable transitions which results in non-deterministic behavior.

**Change Feature Value (CFV)** This operator flips the feature value of a mapping. A modeler may have selected the wrong value for this boolean property of each mapping.

The operator must not be applied to a mapping, if there is a second mapping with the same feature, but different feature value. Otherwise, there will be two mappings for the same feature with the same feature value, which is not allowed for our feature mapping models.

This operator may yield invalid mutants, if it is applied to a mapping that excludes another feature. In that case, two otherwise excluding UML elements can be present at the same time, which may result in invalid models, e.g. two default values assigned to a single variable or concurrently enabled transitions.

### 4.2.2 UML State Machine

In the past 20 years, many mutation operators for transition-based systems were defined [12, 24, 2, 3]. Here, we limit ourselves to the design of operators based on transitions as these may have the strongest impact on the behavior of the SUT. We do not design operators that can be mimicked by the combination of two of them. In particular, we do not consider the exchange of an element by another, since this can easily be mimicked by removing and inserting the removed element at another point in the model.

We identified five targets for mutation: (i) remove the entire transition, change its (ii) target state, as well as mutating its (iii) triggers, (iv) guard, and (v) effect. The latter three can be mutated according to the three defined categories delete, add and change. Though in this contribution omitted the category change for simplicity.

For all mutants created by the here presented operators, there is a chance of materializing mutants behaving equivalent to the original product. This is the case, when the mutated element is part of disabled feature. Of course, hidden mutants – if detected – will be excluded from the scoring.
In general, we will not apply any class mutation to our UML state machines [19]. The system’s logic is designed in the state machine diagrams, while the classes are merely containers for variables and diagrams.

**Delete Transition (DTR)** Deletes a transition from a region in a UML state machine. This operator might create invalid UML models, if not enough transitions remain on a pseudo-state (fork, join, junction, and choice) [22, p.555].

**Change Transition Target (CTT)** Changes the target of a transition to another state of the target state’s region. This operator is only applicable if the region has more than one state.

**Delete Effect (DEF)** Deletes the entire effect from a transition. We consider sending signals to the environment or other components to be part of a transition’s effect, hence they are deleted as well.

**Delete Trigger (DTI)** Deletes a transition’s trigger. Only a single trigger is deleted at a time, but every trigger is deleted once.

**Insert Trigger (ITG)** Copies an additional trigger to a transition. The trigger is copied from another transition within the same region. This may lead to non-deterministic behavior if both transitions, the source transition of the trigger and the mutated transition, are outgoing transitions of the same state.

**Delete Guard (DGD)** Deletes the entire guard of a transition. This may lead to non-deterministic behavior of the state machine, if another transition is enabled simultaneously. Furthermore, in the case of transitions without triggers and where source and target are the same state, this operator leads to infinite looping of the state machine over the mutated transition. The reason for this behavior is UML’s run-to-completion semantic, where an enabled transition without triggers is immediately traversed.

**Change Guard (CGD)** Changes a guard’s term by exchanging operators or substituting boolean literals by their inverse. Our CGD operator supports 30 different arithmetic, relational, bitwise, compound assignment, and logic operators. Furthermore, literal ”null” is exchanged by ”this”. This may cause mutants with non-deterministic behavior, whenever two transition become concurrently enabled due to the manipulation of one of their guards.

### 4.3 Evaluation

We created three example product lines for performing a mutation analysis on them. We designed the test suite for each example automatically by applying model-based testing techniques. In particular, we used product line-centered test design (PLC) from our SPLTestbench as defined in [20], where tests are designed from the SPL specification. In contrast to product-centered test approaches, where tests are designed from selected product specifications, the PLC approach selects products for testing after the test design phase. This improves coverage of the state machine, since the coverage criteria are applied onto the whole SPL specification.

We chose all-transitions coverage for selecting the tests. A test generator then automatically designed the tests and outputs XML-documents. From the tests, SPLTestbench selected variants for testing and materialized them from the mutated SPL specifications into product specification mutants.

Since our examples lack implementations, we decided to generate code from the product specification mutants and run the tests against them. Therefore, we developed and employed a code generator for transforming individual product specifications into Java. Another transformer generates executable JUnit code from the tests which we gained from the test generator. The mutation systems then collects all the code artifacts, executes the tests against the product code, and finally reports the mutation scores
for all tests and for every operator individually. All of the transformations above and the mutation system are part of our SPLTestbench.

Generating code and tests from the same basis for testing the code is not feasible in productive environments, since errors propagate from the basis to code and tests. However in our case, tests are executed against code derived from mutated artifacts, which are different from the original.

4.3.1 Examples

Our examples represent three kinds of systems: an e-commerce shop (eShop), which makes heavy use of signals but with only few guards, a ticket machine (TicketMach) that uses less signals and in contrast more guards, and lastly, an alarm system (AlarmSys), which uses various signals and guards and is more variant-rich than the other two case studies.

The eShop is a fictional example designed by ourselves, which is comprised of 10 features offering 20 different variants. A customer can browse the catalog of items, or if provided, use the search function. Once the customer put items into the cart, he can checkout and may choose from up to three different payment options, depending on the eShop’s configuration. The transactions are secured by either a standard or high security server. A constraint ensures that credit card payment is only offered if the eShop also implements a high security server.

The TicketMach example is adopted from Cichos et al. [7]. The functionality is as follows: a customer may select tickets, pay for them, receive the tickets, and collect change. The feature model has a root feature with three sub-features attached to it; all of them are optional with no further constraints, thus it offers eight variants. Depending on the selected features, the machine offers reduced tickets, accepts not only coins but also bills, and/or will dispense change.

From Cichos et al. [6] we also adopted and extended the AlarmSys example. Currently, it consists of 12 features and offers 42 variants. The alarm may be set off manually or automatically by a vibration detector. Both features are part of an or-group and, thus, at least one of the two features must be present in every product. In the event of an alarm, a siren or a warning light will indicate the security breach. When the vibration does not stop after a predefined period of time, the system optionally escalates the alarm by calling police authorities and/or sending photos of evidence. Additionally to its alarming functionality, the AlarmSys SPL provides a feature for taking a photo of any operator that configures the system for security measures.

4.3.2 Results

We were able to assess the test quality for all three test suites derived from the examples. Here, we present our results. For each mutation operator we measured the amount of detected mutants based on the SPL specification. In addition, we assessed accumulated mutation scores for each example over all mutation operators and vice versa, the accumulated results for each mutation operator over all examples. The detailed results for feature mapping operators can be read from Table 1 and for UML operators from Table 2.

Furthermore, we tracked for every example the number of original products selected for testing, generated product line mutants, and materialized product mutants. Test-wise we counted tests, test steps by means of stimuli and expected reactions in all tests, tests executed against all product mutants, and the number of failed tests during test execution.

For the eShop example, SPLTestbench selected four products for testing. Independent from this, the mutation system generated 30 product line mutants and 96 product mutants for the mapping mutation
operators. For the state machine mutation operators it generated 122 product line mutants and 478 product mutants. Every test from the 13 tests for this example were executed against every suitable mutant. This results in 302 test executions for the mutants created by the mapping mutation operator and 1553 test execution for state machine mutation operators. Ultimately, 20 tests for mapping operators and 283 tests for state machine operators failed, killing 69.67% and 36.67% of the mutants.

Analog to the eShop, we executed less tests and generated less product mutants for the feature mapping operators: 252 tests were executed against 56 product mutants. The tests yield an even lower mutation score of 35.71% than for the eShop case study.

In case of the AlarmSys, we executed 537 tests against 278 product mutants created by the mapping mutation operators and 1168 tests against 585 product mutants created by the state machine mutation operators. Eventually, 37 and 123 tests failed, killing 30.19% and 57.17% of the mutants, respectively. The results are summarized in Table 3 and 4.

5 Related Work

Mutation analysis for SPLs seems to be a rather new topic. To our knowledge, there is no publication dealing with mutation operators on all model artifacts of a SPL specification. Though, Henard et al. proposed two mutation operators for feature models based on propositional formulas in [15]. They employ their mutation system for showing the effectiveness of dissimilar tests, in contrast to similar tests. For calculating dissimilarity, the authors provide a distance metric to evaluate the degree of similarity between two given products.

In contrast, mutation analysis for behavioral system specifications, e.g. finite state machines, is established since two decades. Fabbri et al. introduced mutation operators for finite state machines
in [12]. In addition to our operators, they also consider adding states and the exchange of elements (event, guard, effect) by another. Belli and Hollmann provide mutation operators for multiple formalism: directed graphs, event sequence graphs [2], finite state machines [25], and basic state charts [3]. They conclude, that there are two basic operations from which most operations can be derived: omission and insertion. Also for timed automata, mutation operators can be found in [1].

In [32] Stephenson et al. propose the use of mutation testing for prioritizing test cases from a test suite in a SPL environment. Unfortunately, the authors provide no evaluation of their approach.

6 Conclusions

In this contribution, we lifted mutation analysis to the product line level. We defined and investigated mutation operators for feature models, mapping models, and UML models. As opposed to product-based mutation analysis, our mutation operators are based on the SPL specification. This allows us to mimic realistic errors made by humans during modeling a SPL. To our knowledge, this is the first step towards a qualitative evaluation of SPL tests, which is based on the SPL’s specification.

Our results for the three examples are as expected for most of the mutation operators. As predicted, mutation operators contributing superfluous behavior are hard to detect for conformance tests. Such mutations are DMP (0%) and DME (0%) on feature mappings and ITG (21.67%) on domain models. For most of the other operators we gain scores above 70%, which is in the expected range for all-transitions coverage [33]. For DGD and CTT mutations the tests score surprisingly low results. Here, further investigations seem necessary.

In conclusion, we identified a lack of error detection capability in standard test procedures for SPLs. Even simple errors are not detectable, neither by all-transitions, MC/DC as for safety-critical system, nor any other conformance test procedure. As indicated, the results are applicable to at least the here surveyed SPL engineering paradigms negative/positive variability and delta modeling. We assume, other paradigms suffer from this lack as well. Unfortunately, current procedures for negative testing, which could potentially detect such errors are still not enabled for SPLs. Thus, future work will proceed to enable negative testing procedures for SPLs.

In [20], we described product-centered and product line-centered test design processes for SPLs. We plan to employ this mutation system for assessing the quality of the test suites generated by the different test design methods. From the results we hope to gain general directions towards favorable test design methods and processes by means of error detection capability, test effort, and efficiency.

Furthermore, we want to investigate higher-order mutation operators, that combine more than one change at a time to a product. For this purpose we need co-adaptations, so that the parts that constitute the SPL, here feature model, mapping/delta model, and domain model, are adapted to preserve consistency when one of the parts changes. For example, such an adaptation is necessary after the deletion of a feature to ensure that there is no broken feature reference in related mappings. In [31], we presented a prototype for co-adaptations in another model-based scenario, namely domain-specific language development.

Acknowledgments

This work is supported by grants from Deutsche Forschungsgemeinschaft, Graduiertenkolleg METRIK (GRK 1324).
References


[33] Stephan Weißleder (2009): Influencing Factors in Model-Based Testing with UML State Machines: Report on an Industrial Cooperation. In David Hutchison et al., editor: Model Driven Engineering Languages and
Adaptive Homing is in P

Natalia Kushik
Tomsk State University
Tomsk, Russia
ngkushik@gmail.com

Nina Yevtushenko
Tomsk State University
Tomsk, Russia
yevtushenko@sibmail.com

Homing preset and adaptive experiments with Finite State Machines (FSMs) are widely used when a non-initialized discrete event system is given for testing and thus, has to be set to the known state at the first step. The length of a shortest homing sequence is known to be exponential with respect to the number of states for a complete observable nondeterministic FSM while the problem of checking the existence of such sequence (Homing problem) is PSPACE-complete. In order to decrease the complexity of related problems, one can consider adaptive experiments when a next input to be applied to a system under experiment depends on the output responses to the previous inputs. In this paper, we study the problem of the existence of an adaptive homing experiment for complete observable nondeterministic machines. We show that if such experiment exists then it can be constructed with the use of a polynomial-time algorithm with respect to the number of FSM states.

1 Introduction

Finite State Machines (FSMs) are widely used when deriving high quality tests for reactive discrete event systems. If a system is non-initialized then homing and synchronizing experiments with FSMs are used in order to set the system into the known state [13]. Homing experiments can be preset and adaptive when the next input significantly depends on output responses produced to the previously applying inputs. As the underlying model for synchronizing experiments is a finite automaton without outputs, only preset synchronizing experiments are considered in various papers (see, for example [15], [2], [6]).

Homing experiments are well studied for deterministic FSMs where minimal length homing sequences are derived based on a truncated successor tree [4], [7]. Any deterministic complete reduced FSM with \( n \) states has a homing sequence of length up to \( n(n - 1)/2 \). A related detailed survey on deriving synchronizing sequences is given by Sandberg in [13]. As usual, when performing 'gedanken' experiments with state models, a transition behaviour of the machine under experiment is supposed to be known [11].

Nowadays, nondeterministic FSMs are thoroughly studied for deriving tests with the guaranteed fault coverage. The reason is that the specification FSM can be nondeterministic according to a number of reasons. For example, it can be a corollary to the optionality, as it happens when extracting an FSM from corresponding RFC specifications [16], or non-determinism can occur according to the limited controllability and/or observability when testing a component of a modular system [3]. Correspondingly, there are a number of publications about test derivation against nondeterministic FSMs. An FSM is nondeterministic if at some state the machine has several transitions under a given input. If an FSM under test is non-initialized a homing preset or adaptive sequence has to be applied before a test sequence in order to set the FSM into the known state.

For preset homing experiments for nondeterministic FSMs, Kushik et al. [8] show that differently from deterministic FSMs a homing sequence does not necessarily exist for a complete reduced nondeterministic FSM and proposed an algorithm for deriving a preset homing sequence for a given observable.
nondeterministic FSM when such sequence exists. A tight upper bound on a shortest preset homing sequence is exponential with respect to to the number of FSM states while the problem of checking the existence of a preset homing sequence for nondeterministic FSMs (Homing problem) is PSPACE-complete [10].

In order to decrease the complexity of related problems, adaptive homing experiments can be used when a next input that is applied to a machine under experiment is selected based on the output responses to previously applied inputs. However, Hibbard [5] showed that, in general, for deterministic machines adaptive homing experiments do not shorten the length of an applied input sequence. In other words, deterministic complete machines require adaptive homing experiments with the height of the same order as for the preset case. However, it is not the case for nondeterministic FSMs. For nondeterministic FSMs, the upper bound on the height of adaptive homing experiments and the complexity of checking the existence of a homing adaptive experiment can be reduced with respect to the preset case. In this paper, we study the problem of checking the existence of an adaptive homing experiment for a nondeterministic FSM and refer to this problem as an “Adaptive Homing” problem.

Similar to Agrawal et al. [1] who first showed the existence of an unconditional polynomial-time algorithm for checking if an integer is prime or composite, we present an approach for checking and deriving (if possible) an adaptive homing test case that represents an adaptive homing experiment for a given nondeterministic FSM. The unconditional polynomial-time algorithm is based on efficient checking of the existence of a homing test case for each pair of the FSM states. The complexity of this algorithm significantly depends on the number of FSM inputs and outputs as well as on the number of FSM states. Assuming that the number of FSM inputs as well as the number of its outputs is polynomial with respect to the number of states, we prove the polynomial complexity of an Adaptive Homing problem.

The structure of the paper is as follows. Section 2 contains Preliminaries. In Section 3, an approach for checking the existence of a homing test case for a given pair of FSM states is presented and the complexity of the Adaptive Homing problem is evaluated. Section 4 concludes the paper.

2 Preliminaries

A (non-initialized) Finite State Machine (FSM) \( S \) is a 5-tuple \( (S, I, O, h_S) \), where \( S \) is a finite set of states; \( I \) and \( O \) are finite non-empty disjoint sets of inputs and outputs; \( h_S \subseteq S \times I \times O \times S \) is a transition relation, where a 4-tuple \( (s, i, o, s') \in h_S \) is a transition.

An FSM \( S = (S, I, O, h_S) \) is complete if for each pair \( (s, i) \in S \times I \) there exists a pair \( (o, s') \in O \times S \) such that \( (s, i, o, s') \in h_S \); otherwise, the machine is partial. Given a partial FSM \( S \), an input \( i \) is a defined input at state \( s \) if there exists a pair \( (o, s') \in O \times S \) such that \( (s, i, o, s') \in h_S \). FSM \( S \) is nondeterministic if for some pair \( (s, i) \in S \times I \), there exist at least two transitions \( (s, i, o_1, s_1) \), \( (s, i, o_2, s_2) \in h_S \), such that \( o_1 \neq o_2 \) or \( s_1 \neq s_2 \). FSM \( S \) is observable if for each two transitions \( (s, i, o_1, s_1) \), \( (s, i, o_2, s_2) \in h_S \) it holds that \( s_1 = s_2 \). FSM \( S \) is single-input if at each state there is at most one defined input at the state, i.e., for each two transitions \( (s, i_1, o_1, s_1) \), \( (s, i_2, o_2, s_2) \in h_S \) it holds that \( i_1 = i_2 \), and FSM \( S \) is output-complete if for each pair \( (s, i) \in S \times I \) such that the input \( i \) is defined at state \( s \), there exists a transition from \( s \) with \( i \) for every output in \( O \) [12]. The FSM with the designated initial state \( s_0 \) is an initialized FSM, written \( (S, s_0, I, O, h_S) \). Given initialized FSMs \( S = (S, s_0, I, O, h_S) \) and \( P = (P, p_0, I, O, h_P) \), the FSM \( P \) is a submachine of \( S \) if \( P \subseteq S \), \( p_0 = s_0 \) and \( h_P \subseteq h_S \). A trace of \( S \) at state \( s \) is a sequence of input/output pairs of sequential transitions starting from state \( s \). Given a trace \( (i_1, o_1) \ldots (i_k, o_k) \) at state \( s \), the input projection \( i_1 \ldots i_k \) of the trace is a defined input sequence at state \( s \). An initialized FSM \( S = (S, s_0, I, O, h_S) \)
is acyclic if the set $Tr(S/s_0)$ of traces at the initial state is finite, i.e., the FSM transition diagram has no cycles. As usual, for state $s$ and a trace $\gamma$, the $\gamma$-successor of state $s$ is the set of all states that are reached from $s$ by $\gamma$. If $\gamma$ is not a trace at state $s$ then the $\gamma$-successor of state $s$ is empty or we simply say that the $\gamma$-successor of state $s$ does not exist. For an observable FSM $S$, the cardinality of the $\gamma$-successor of state $s$ is at most one for any trace $\gamma$. Given a nonempty subset $S'$ of states of the FSM $S$ and a trace $\gamma$, the $\gamma$-successor of the set $S'$ is the union of $\gamma$-successors over all $s \in S'$.

As in this paper we consider homing experiments with nondeterministic FSMs, in order to identify a state of a given non-initialized FSM after the experiment, a finite input sequence is applied to the FSM where the next input (except of the first one) of the sequence is determined based on the output of the FSM produced to the previous input. Formally, such an experiment can be described using a single-input output-complete FSM with an acyclic transition graph and similar to [12] we refer to such an FSM as a test case. A test case $P$ is homing for the set $S'$ of states of the FSM $S$ if for each trace $\gamma$ from the initial state to a deadlock state of $P$, there exists a state $s$ of $S$ such that for each state $s' \in S'$, the $\gamma$-successor of state $s'$ does not exist or the $\gamma$-successor of state $s'$ is $s$. If there exists a homing test case for the set $S'$ then the set $S'$ is adaptively homing or simply a homing set. If there exists a homing test case for the set $S$ of all states then FSM $S$ is adaptively homing.

Given an input alphabet $I$ and an output alphabet $O$, a test case $TC(I,O)$ is an initially connected single-input output-complete observable initialized FSM $P = (P, I, O, h_P, p_0)$ with the acyclic transition graph. By definition, if $|I| > 1$ then a test case is a partial FSM. A test case $TC(I,O)$ over alphabets $I$ and $O$ defines an adaptive experiment with any complete FSM $S$ over the same alphabets. A test case $TC(I,O)$ over alphabets $I$ and $O$ is a homing test case for FSM $S = (S, I, O, h_S)$, if for each trace $\gamma$ of the test case from the initial state to a deadlock state the $\gamma$-successor of the set $S$ is a singleton.

3 Adaptive Homing problem for a pair of FSM states

An FSM is adaptively homing [9] if there exists an adaptive homing experiment. In general, given a test case $P$, the length of the test case $P$ is determined as the length of a longest trace from the initial state to a deadlock state of $P$ and it specifies the length of the longest input sequence that can be applied to an FSM $S$ during the experiment that is also often called the height of the adaptive experiment. In this section, we discuss how an Adaptive Homing problem can be solved for a given pair of FSM states. This problem is stated as follows. Given a complete nondeterministic FSM $S = (S, I, O, h_S)$, and a pair $(s_i, s_j)$, the question is whether there exists a homing test case for a pair $(s_i, s_j)$. As the reply, there can be a homing test case for a pair $(s_i, s_j)$ or a message “the pair $(s_i, s_j)$ is not adaptively homing”.

In this section, we focus on solving Adaptive Homing problem for a pair of FSM states, since if the FSM under experiment is observable then Adaptive Homing problem for FSM $S$ can be reduced to this problem for each state pair. Given a state pair $(s_i, s_j)$, we propose a procedure for checking whether this pair of states is adaptively homing based on the corresponding FSM intersection. The complexity evaluation seems to be more straightforward when using the corresponding intersection than for the procedure proposed in [9]. Without loss of generality, consider a pair $(s_1, s_2)$ of FSM states.

Given a complete observable FSM $S = (S, I, O, h_S)$ and two different states $s_1$ and $s_2$ of $S$, we derive the intersection $S/s_1 \cap S/s_2 = (Q, (s_1, s_2), I, O, h_S, s_{(s_1, s_2)})$ in a usual way. States of $S/s_1 \cap S/s_2$ are pairs $(s_j, s_k)$, $j < k$, $j, k = 1, \ldots, n$, and there is a transition $((s_j, s_k), i, o, (s', s'))$ if and only if $s_j \neq s_k$ and $s'_j, s'_k$ are io-successors of states $s_j$ and $s_k$. If for all $o$ there are no such io-successors then a transition at the state $(s_j, s_k)$ under input $i$ is not defined.

**Proposition 3.1** Given two states $s_1$ and $s_2$ of a complete observable FSM $S = (S, I, O, h_S)$ and the
intersection $S/s_1 \cap S/s_2$, states $s_1$ and $s_2$ are not adaptively homing if and only if the intersection $S/s_1 \cap S/s_2$ has a complete submachine.

**Proof** $\Leftarrow$ Let there exist a complete submachine $F$ of the intersection $S/s_1 \cap S/s_2$. By definition, this means that for every input sequence $\alpha$ there exists an output sequence $\beta$ such that the trace $\alpha/\beta$ takes the pair $(s_1,s_2)$ to another pair $(s_j,s_k)$, $s_j \neq s_k$, i.e., the pair $(s_1,s_2)$ is not adaptively homing.

$\Rightarrow$ Consider states $s_1$ and $s_2$ which are not adaptively homing. Derive a subset $Q'$ of states of FSM $Q = S/s_1 \cap S/s_2$ which are not adaptively homing. For each state $q \in Q'$ and each input $i \in I$, there exists a transition $(q,i,o,q')$ in $Q$ for some $o \in O$ and $q' \in Q$ and moreover, at least one of such states $q'$ is in the set $Q'$; otherwise, state $q$ is adaptively homing. Since the initial state of is not adaptively homing, $(s_1,s_2) \in Q'$; and thus, the FSM $Q$ has a complete submachine with the set $Q'$ of states.

**Corollary 3.2** States $s_1$ and $s_2$ are adaptively homing if and only if the intersection $S/s_1 \cap S/s_2$ has no complete submachine, i.e., each submachine has an input undefined in some state.

**Proposition 3.3** The Adaptive Homing problem for given two states $s_1$ and $s_2$ of a complete observable FSM $S = (S,I,O,h_S)$ is in $P$, when the number of FSM inputs/outputs is polynomial w.r.t. the number of FSM states.

**Proof** In order to estimate the complexity of the Adaptive Homing problem for two states $s_1$ and $s_2$, we evaluate the corresponding complexity as a function of the number $n = |S|$ of FSM states. The existence of a complete submachine can be checked by iterative removal from the intersection $S/s_1 \cap S/s_2$ each state that has an undefined input along with its incoming transitions. If at the end, the initial state is also removed then the two given states are adaptively homing, otherwise they are not adaptively homing. The procedure is polynomial with respect to the number of states [14] when the number of inputs and outputs are polynomial with respect to the number of states, and thus, the complexity of checking the existence of a complete submachine of $S/s_1 \cap S/s_2$ is polynomial, since this machine has at most $n(n-1)/2$ states.

We mention, that the procedure of checking whether the pair of states is homing, given in Proposition 2, is similar to the one for checking the existence of an adaptive $(s_1,s_2)$-distinguishing strategy considered in [12]. Once the existence of a homing test case is proven one can derive such test case in various ways. A reader may turn to [12] where an algorithm for deriving an adaptive distinguishing test case is proposed or to [9] where a general procedure for deriving a homing test case for a weakly initialized FSM is proposed. In both cases, the number of states of a shortest test case for a pair of FSM states is at most $n(n-1)/2 + 1$, where the integer 1 is added for the designated deadlock state. As a test case is a single-input output complete FSM, the maximal number of the test case transitions equals the product of $(n(n-1)/2)$ and $|O|$. In other words, the following proposition holds.

**Proposition 3.4** Given a complete observable nondeterministic FSM $S$ and states $s_1$ and $s_2$ of FSM $S$, if states $s_1$ and $s_2$ are adaptively homing then the number of transitions of a shortest homing test case does not exceed $(n(n-1)/2) \cdot |O|$.

**Corollary 3.5** Given a complete observable adaptively homing nondeterministic machine $S$, a shortest homing test case requires a polynomial size of memory for its storage, if the number of FSM outputs is polynomial with respect to the number of its states.

In [9], the following statement is established.

**Proposition 3.6** [9] A complete observable FSM $S$ is adaptively homing if and only if each pair of two different states is homing.
Since for an FSM with \( n \) states, the number of pairs of different states is \( n(n - 1)/2 \), the above proposition immediately implies the following statement.

**Proposition 3.7** The problem of checking of the existence of a homing test case for a complete observable FSM \( S \) is in \( P \).

**Example.** Consider an FSM with a flow table in Table 1. By direct inspection, one can assure that each pair of states is adaptively homing. A flow table for the intersection \( S/s_1 \cap S/s_2 \) that has no complete submachine, is shown in Table 2. States 1 and 3 can be homed by the input \( i_2 \) while States 2 and 3 can be homed by the input \( i_1 \).

By direct inspection, one can assure there does not exist a complete deterministic submachine for the FSM \( S/s_1 \cap S/s_2 \) with a flow table in Table 2. Therefore, the FSM \( S \) with a flow table in Table 1 is adaptively homing. One of homing test cases for this machine is shown in Table 3.

### 4 Conclusion

In this paper, we have shown that the problem of checking the existence of a homing test case for a given complete observable FSM is in \( P \). Moreover, we have shown that for a pair of states of an adaptively homing FSM a shortest homing test case requires a polynomial size of memory for its storage, if the number of FSM outputs is polynomial with respect to the number of its states.

Keeping in mind that the procedure for deriving a homing test case for the set of all states of a complete observable FSM asks in fact for concatenation of test cases for pairs of states, we can presume that for an adaptively homing FSM, a shortest homing test case requires a polynomial size of memory for its storage when the number of FSM outputs is polynomial with respect to the number of its states. Nevertheless, we leave the proof of this statement for the future work as well as the study whether the problem of deriving a shortest homing test case is also in \( P \). Another interesting problem can be the exact

<table>
<thead>
<tr>
<th>Input/State</th>
<th>1, 2</th>
<th>2, 3</th>
<th>1, 3</th>
<th>( p )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( i_1 )</td>
<td>1, 3/( o_1 ), 2, 3/( o_2 )</td>
<td></td>
<td>1, 3/( o_1 )</td>
<td>( p/( o_1 ), ( o_2 )</td>
</tr>
<tr>
<td>( i_2 )</td>
<td></td>
<td></td>
<td>1, 3/( o_1 )</td>
<td>( p/( o_1 ), ( o_2 )</td>
</tr>
<tr>
<td>( i_3 )</td>
<td>1, 2/( o_1 )</td>
<td>1, 3/( o_2 )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
characterization of all homing test cases in the form of an FSM, called a canonical homing test case, similar to the canonical separator for distinguishing test cases in [12].

References