Heuristic-driven Techniques for Test Case Selection

J.C. Burguillo, M. Llamas, M.J. Fernández 1

Departamento de Ingeniería Telemática
Universidad de Vigo
Vigo, Spain

T. Robles 2

Departamento de Ingeniería Telemática
Universidad Politécnica de Madrid
Madrid, Spain

Abstract

We propose an approach to testing that combines formal methods with practical criteria, close to the testing engineer’s experience. It can be seen as a framework to evaluate and select test suites using formal methods, assisted by informal heuristics. We also introduce the formalism of enriched transition systems to store information obtained during the testing phase, and to adapt classical test generation techniques to take advantage of the possibilities of the new formalism.

1 Introduction

In the context of Protocol Engineering, test generation algorithms are used to obtain a set of test cases from a given specification, intended to detect errors in non-conforming implementations. However, the number of test cases needed to guarantee an exhaustive coverage may be too large, even infinite. Therefore, execution of all potential test cases may be infeasible. As a consequence, in practical cases it is necessary to select a subset of all possible test cases prior the test execution phase. The reduction of the initially generated test case set is known in the literature as test case selection.

Test case selection should not be performed at random. An appropriate strategy should be applied to obtain a valuable test case collection, in the sense

1 Email: [jrial,martin,manolo]@det.uvigo.es
2 Email: robles@dit.upm.es

©2002 Published by Elsevier Science B. V.
that it should detect as many non-conforming implementations as possible. For software testing, some criteria are available, like the division in equivalence partitions [12] or the test proposal selection in protocol testing [9].

On the other side, test case selection should not be based only on the system’s formal specification. To select the most valuable test cases, additional information, external to the corresponding specification formalism, should also be used. Such information may consider most frequent errors committed by implementors, most harmful errors, most difficult to implement features, critical basic functionalities, etc.

In the field of Formal Description Techniques some proposals have been made to address the test case selection problem, key results may be found in [6,15,16,17]. T. Robles [13] introduced concepts for risk, cost and efficiency for a test case collection, which are revisited in this paper. This approach is based on the estimation, from the testing engineer’s experience, of the risk involved when testing a system implementation. It formalises and simplifies the selection of test cases, and can be applied to most practical problems. This approach is similar to that presented in [15].

Thus, this paper proposes a method to evaluate and select test cases from practical criteria, close to the testing engineer’s experience. Our aim is to provide implementable, and computationally feasible criteria. Additionally, we want the proposed methodology to be easily configurable for testing engineers, who can provide their experience through the introduction of heuristics to facilitate the testing of key aspects in a system, or specific parts of a system that are more prone to errors.

The next two sections discuss the theoretical background that serves as the foundation of our experience. Section 2 presents some general definitions and notation about the supporting representation framework and formal testing, and Section 3 presents our approach to test case selection. Finally, Section 4 offers a summary of the work described and some conclusions.

2 General Definitions and Notation

Along the next paragraphs we discuss basic theoretical concepts and notation related to testing and test case selection. First, we briefly introduce Labelled Transition Systems. Then, we provide some basic concepts from formal testing. After this, we introduce risk, coverage, cost and efficiency as the supporting heuristics to assist the testing engineer along test case selection.

2.1 Labelled Transition Systems

Labelled Transition Systems (LTS) will be the basic model to describe the behaviour of processes, such as specifications, implementations and tests.

Definition 1 A labelled transition system is a 4-tuple $<\text{Stat},L,T,s_0>$ where $\text{Stat}$ is a countable, non-empty set of states; $L$ is a countable set of
labels; $T \subseteq \text{Stat} \times (L \cup \{i\}) \times \text{Stat}$ is the countable set of transitions and $i$ denotes a special internal action, referred as $\tau$ in some models [11]; and $s_0 \in \text{Stat}$ is the initial state.

An element $(s, \mu, s') \in T$ is represented as $s-\mu \rightarrow s'$. We use the following notations (sets) derived (constructed) from the transition relation:

- $s = \epsilon \Rightarrow s'$: $s = s'$ or $s - i - \ldots \rightarrow s'$
- $s = a \Rightarrow s'$: $\exists s_1, s_2 \in \text{Stat}$ such that $s = \epsilon \Rightarrow s_1 - a \rightarrow s_2 = \epsilon \Rightarrow s'$
- $s = \sigma \Rightarrow s'$: $\exists\{s_1, \ldots, s_{n-1}\} \subseteq \text{Stat}$, and a trace $\sigma = a_1 \ldots a_n$
  such that $s = a_1 \Rightarrow s_1 = \cdots \Rightarrow s_{n-1} = a_n \Rightarrow s'$.
- $s = \sigma \Rightarrow$: $\exists s' \in \text{Stat}$ such that $s = \sigma \Rightarrow s'$
- $s \neq \sigma \Rightarrow$: $\forall s' \in \text{Stat}$ such that $s = \sigma \Rightarrow s'$

$\text{Tr}(P)$: $\{\sigma \in L^* \mid P = \sigma \Rightarrow\}$
$\text{Init}(P)$: $\{a \in L \mid P = a \Rightarrow\}$
$P \text{ after } \sigma$: $\{s' \mid P = \sigma \Rightarrow s'\}$
$\text{Ref}(P, \sigma)$: $\{A \subseteq L \mid \exists s' \in (P \text{ after } \sigma) \text{ and } \forall a \in A, s' \neq a \Rightarrow\}$
$\text{Path}(P)$: $\{\varphi \in T^* \mid P - \varphi \rightarrow s', s' \in \text{Stat}\}$

The symbol $L^*$ (respectively $T^*$) denotes the set of strings (sequences, traces) constructed using elements from $L$ (respectively $T$). A trace $\sigma \in L^*$ is a finite sequence of observable actions over $L$, where $\epsilon$ denotes the empty sequence. The special label $i \notin L$ represents an unobservable, internal action, used to model non-determinism. Thus $= \epsilon \Rightarrow$ represents a null transition or a sequence of transitions including only internal actions (i.e. traces do not have internal actions). We use $t \ll \varphi$ to denote that transition $t$ appears in the path $\varphi$.

We represent an LTS by a tree or a graph, where nodes represent states and edges represent transitions. Given an LTS $P =< \text{Stat}, L, T, s_0 >$, we write $P = \sigma \Rightarrow$ to represent transitions from the initial state of $P$ and must be considered as a syntax sugar. When a given state does not accept further actions (i.e. deadlock state), we label it as $\text{stop}$.

$\text{Tr}(P)$ is the set of traces accepted by process $P$, $\text{Init}(P)$ the set of labels from $L$ accepted by $P$, and $\text{Ref}(P, \sigma)$ is the set of refusals of $P$ after trace $\sigma$. Finally, $\text{Path}(P)$ is the set of transition sequences accepted by $P$. We denote the class of all labelled transition systems over $L$ by $\text{LTS}(L)$. LTS model the semantics of languages used to describe distributed and concurrent systems like LOTOS [8], CSP [1] or CCS [11], among others.
2.2 Formal Testing Concepts

Concerning testing, it is important to define a relation to model the conformance of an implementation with its specification. There are several relations in the literature that may be selected [14]. As we want to compare our framework with other approaches and reuse the existing theory, we selected the conformance relation $\text{conf}$ described in [2,14]. It has the advantage that only the behaviour contained in the specification must be tested, reducing the test space. The relation $\text{conf}$ is defined as follows:

**Definition 2 (Conformance: $\text{conf}$)** Let $I, S \in \text{LTS}(L)$, we say that $I \text{conf} S$ if and only if for every trace $\sigma \in \text{Tr}(S)$ and for every subset $A \subseteq L$ the following proposition holds: If $A \in \text{Ref}(I, \sigma)$ then $A \in \text{Ref}(S, \sigma)$

In case $\sigma \notin \text{Tr}(I)$ we assume $\text{Ref}(I, \sigma)$ is empty.

To decide about the success of a test case we use *verdicts*. Reference [10] proposes three possible verdicts: **Pass** ($\text{pass}$, when the observed behaviour satisfies the test), **Fail** ($\text{fail}$, when the observed behaviour is an invalid specification behaviour) and **Inconclusive** ($\text{inc}$, the observed behaviour is valid so far, but it has not been possible to complete the test). These concepts are formalised below [14]:

**Definition 3 (Test case)** A test case $tc$ is a 5-tuple $< \text{Stat}, L, T, v, s_0 >$, such that $< \text{Stat}, L, T, s_0 >$ is a deterministic transition system with finite behaviour, and $v : \text{Stat} \rightarrow \{\text{fail, pass, inc}\}$ is a function to assign verdicts.

**Definition 4 (Test suite)** A test suite or test collection $ts$ is a set of test cases: $ts \in \text{PowerSet}(\text{LTS}(L))$

The execution of a test case is modelled by the parallel synchronous execution of the test case with the implementation under test (IUT). Such execution continues until there are no more interactions, i.e. a deadlock is reached. Such deadlock may appear because the test case $tc$ reaches a final state, or when the combination of $tc$ and the IUT reaches a state where the actions offered by $tc$ are not accepted.

An implementation passes the execution of a test case if and only if the verdict of the test case is **pass** when reaching a deadlock. As the implementation may have nondeterministic behaviour, different executions of the same test case with the same IUT may reach different final states, and as a consequence different verdicts. An implementation passes a test case $tc$ if and only if all executions of $tc$ produce a **pass** verdict. This means that we should execute every test case several times to obtain a final verdict, ideally an infinite number of times.

Test generation algorithms provide test suites from specifications. Ideally, an implementation must pass a test suite if and only if it conforms. Unfortunately, in practice, such test suite would have infinitely many test cases. As a consequence, in the real world we have to restrict ourselves to (finite-size) test suites that can only detect non-conformance, but cannot detect conformance.
Table 1  

Error Weighting

<table>
<thead>
<tr>
<th>Target</th>
<th>Parameter</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event</td>
<td>$R_I(e) = E_I(e) \times I_I(e)$</td>
<td>$(0, \infty)$</td>
</tr>
<tr>
<td>Implementation</td>
<td>$R_I(S)$</td>
<td>$(0, \infty)$</td>
</tr>
<tr>
<td>Measurement, Event</td>
<td>$MR_I(e, ts)$</td>
<td>$[0, \infty)$</td>
</tr>
<tr>
<td>Measurement, Implementation</td>
<td>$MR_I(S, ts)$</td>
<td>$[0, \infty)$</td>
</tr>
</tbody>
</table>

**Legend.** $I$: implementation under test; $e$: event in $I$; $ts$: test suite; $S$: specification corresponding to $I$.

Such test suites are called *sound*.

2.3 Risk, Coverage, Cost and Efficiency

Through the next few paragraphs we introduce the concepts of error weight or risk, coverage, cost and efficiency, which will support the comparison and selection of test cases to be passed to an implementation.

To analyse the coverage obtained after testing an implementation we have to take into account several factors. On one side, test cases are derived from a formal object, i.e. the formal specification. As a consequence, after testing an implementation we get a specific coverage level for the behaviours in the specification. On the other side, coverage depends on the implementation itself because, given a formal specification, the selected implementation technology (i.e. programming language or programming tools) will be more or less prone to errors.

Table 1 proposes some heuristics to *a priori* evaluate the influence of errors in a given implementation, which will be used to select an adequate test suite. $R_I(e)$ assigns a weight to a (possible) error, i.e. estimates the risk involved in committing errors when implementing event $e$. It is calculated from two values: an estimation of the chances of event $e$ being erroneously implemented ($E_I(e)$), and an estimation of the impact of the corresponding error in the rest of the system ($I_I(e)$). $R_I(S)$ estimates the chances for the implementation not to conform to the corresponding specification, and measures the risk of erroneously implementing $S$.

$MR_I(e, ts)$ represents the amount of risk for event $e$ that can be detected through a testing process using test suite $ts$, and $MR_I(S, ts)$ represents the amount of risk for implementation $I$ that can be detected using test suite $ts$. Risk measurement for a single test case is a particular case where suite $ts$ is composed by a single test case. Note that, from the definitions above, $MR_I(e, ts) \leq R_I(e)$ and $MR_I(S, ts) \leq R_I(S)$.

The underlying mathematical model we need is considerably simplified through the assumption of independence among errors. However, in prac-
Burguillo, Llamas, Fernández, Robles

Table 2
Coverage Parameters

<table>
<thead>
<tr>
<th>Target</th>
<th>Parameter</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event</td>
<td>$K_I(e, ts) = \frac{MR_I(e, ts)}{R_I(e)}$</td>
<td>$[0, 1]$</td>
</tr>
<tr>
<td>Implementation</td>
<td>$K_I(S, ts) = \frac{MR_I(S, ts)}{R_I(S)}$</td>
<td>$[0, 1]$</td>
</tr>
</tbody>
</table>

Table 3
Cost Parameters

<table>
<thead>
<tr>
<th>Target</th>
<th>Parameter</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event</td>
<td>$C_I(e) = P_I(e) + X_I(e)$</td>
<td>$(0, \infty)$</td>
</tr>
<tr>
<td>Implementation</td>
<td>$C_I(S, ts)$</td>
<td>$(0, \infty)$</td>
</tr>
</tbody>
</table>

tice, errors are not independent from each other, as erroneous sentences in a program may affect the evolution of other parts of the program. As a solution, correlation among errors is reflected in our model as error weight values, that is, we model such interdependence through parameter $I_I(e)$. Then, testing engineers will estimate the correlation among errors, using available error statistics and their own expertise, to define $I_I(e)$ accordingly.

This can be seen as a compromise between a convenient mathematical foundation and the need to consider error correlation in real cases. Note that, independently of being supported by the underlying mathematical model or through explicit parameters, getting the correlations between failures right is crucial to get the most of the approach discussed in this paper.

From the parameters above, we define coverage as the quotient between a measurement of the detection power of a test suite and a measurement of the risk (c.f. table 2). $K_I(e, ts)$ represents the coverage for event $e$ using test suite $ts$, whereas $K_I(S, ts)$ represents the coverage for implementation $I$, corresponding to specification $S$, using test suite $ts$.

When executing a test suite $ts$ on an IUT we are checking whether some of the error possibilities estimated have been materialised into actual errors. If errors appear, they should be corrected. Conversely, if errors are not found, our confidence increases. Given two test suites $ts_1$ and $ts_2$, using the parameters above we can compare their coverage, and therefore their ability to detect errors in an IUT. However, there is another factor when comparing test suites that should be taken into account: the resources needed. To estimate this aspect, we introduce a new parameter: the cost (c.f. table 3). $C_I(e)$ estimates the cost of testing event $e$ as the sum of the cost due to its implementation in a test case ($P_I(e)$) and the cost of executing that event on the implementation ($X_I(e)$). $C_I(S, ts)$ represents the cost of testing an implementation $I$ using test suite $ts$ generated from specification $S$.

Using cost values we can better discriminate among several test suites. Therefore, the next step will be to relate the parameters defined above to
obtain another reference to facilitate the selection of test cases. For this, we define the efficiency of a test suite ts obtained from \( S \) \( (F_I(S,ts)) \) as the quotient between the coverage of that suite and the cost associated to its use to test \( I \).

\[
F_I(S,ts) = \frac{K_I(S,ts)}{C_I(S,ts)}
\]

The values of this new parameter are in the range \([0, \infty)\). Its values increase when coverage increases and with cost reduction.

We need a procedure to calculate values for the heuristics above taking into account our representation formalism, namely Labelled Transition Systems. We try to assess conformance for a system implementation from its formal specification. Thus, we will take as a reference the risk involved when implementing all events in the specification. In this way, we can formulate the risk for a IUT as the sum of the risk values for its events.

![Fig. 1. S, I₁ and I₂](image)

On the other side, due to nondeterminism, practical test cases should be executed several times to gain confidence on the testing process. For example, consider the specification \( S \) in figure 1 and its implementations \( I_1 \) and \( I_2 \). While the implementation \( I_1 \) is equal to \( S \) and will always accept event \( a \) as stated by \( S \), implementation \( I_2 \) sometimes executes an internal action and then refuses event \( a \). Obviously, this latter implementation does not conform with \( S \).

If we are testing a physical implementation, which may behave as \( I_1 \) or \( I_2 \), we will need to execute several times \( a \) from the initial state in order to discover if it conforms with \( S \). Each time event \( a \) is accepted we increase our confidence on the implementation. Conversely, if we obtain a single refusal we
can guarantee that the IUT does not conform. In other words, measurement risk values vary along the testing process.

Additionally, the presence of recursive behaviours makes testing dependent on the level of recursion where the test is passed. We name recursive behaviours those ones that are self-instantiated. Consequently, the recursion level will be the number of times a behaviour has been instantiated. For instance, specification $S$ in Figure 2 contains a recursive behaviour and never stops. Again, to check a physical implementation of $S$ that behaves as $I$ in Figure 2, we might need to execute many times event $send$ to detect that sometimes such event is refused. As a consequence, the risk measurement involved when testing an event is spread along the successive levels of recursion (i.e. successive event instantiations).

Taking into account both aspects, we can decompose the risk of every event in an LTS (i.e. the weight assigned to errors in events) as:

$$R_I(e) = \sum_{r=1}^{\infty} \sum_{n=1}^{\infty} R_{r,n}^I(e) \leq \infty$$

where $R_{r,n}^I(e)$ represents the risk of event $e$ when being tested for the $n$-th time at recursion level $r$ using a given test case. Then, the risk detection power of a test suite $ts$ becomes:

$$MR_I(S,ts) = \sum_{tc \in ts} \sum_{e \in E(tc)} \sum_{r=1}^{Rc_e} \sum_{n=0}^{N_e(r)} R_{r,n}^I(e)$$

where $Rc_e$ and $N_e(r)$ are respectively the deepest recursion level where event $e$ has been tested and the number of times we tested such event for every recursion level. If test cases $tc \in ts$ have a tree structure we can obtain several possible values for every successful run of the test case. So, we may measure the risk, a priori, using available statistics.

2.4 A Priori and a Posteriori Values

As the IUT is an entity whose behaviour is unknown, there may be differences between what we desire to test and what we really test in practice. These differences may be due to:

- **Nondeterminism**: due to nondeterministic behaviour in the implementation, it is possible that, in a first try, we cannot test those behaviours we are interested in. Because of this, it may be needed to execute test cases several times until we reach an appropriate result. New executions modify coverage values.

- **Failures**: if we detect a non-conforming implementation, it may not be possible to achieve the expected coverage because some test cases may not be executable due to errors in the implementation.

As a consequence we can identify [7] two classes of cost and coverage values:
• **A priori values**, which are obtained when we estimate the risk measurement and the cost to execute a test case $tc$ assuming all possible implementation responses, as defined by the corresponding specification.

• **A posteriori values**, which are obtained after executing the test case $tc$.

### 3 Test Case Selection

Now, we will discuss our approach to test case selection, which is based on a classical approach, as discussed below. But first we introduce Enriched Transition Systems as a way to keep track of the structural information needed to know those parts of the specification already tested.

#### 3.1 Enriched Transition Systems

When we try to execute several test cases over an implementation, it would be desirable to have access to the values of risk, cost and coverage obtained along the process. For this, as discussed above, we need information about recursion levels and testing iterations. Besides, if these values were available, we could select new test cases depending on the results obtained from the ones that have been already executed.

To maintain the information gathered after the execution of test cases we define a new type of transition systems [5]:

**Definition 5 (Enriched Transition System)** An enriched transition system (ETS) is a 5-tuple denoted by $S = \langle \text{Stat}, L, T, N(t, r), s_0 \rangle$, such that $\langle \text{Stat}, L, T, s_0 \rangle$ is a labelled transition system and $N(t, r)$ is the number of times transition $t \in T$ is executed at recursion level $r \in [1, \infty)$.

The set of enriched transitions systems over the label set $L$ is denoted by $ETS(L)$. Available notation and definitions for $LTS(L)$ are extended to $ETS(L)$ defining them over the underlying transition system. Unlike classical LTS, ETS are dynamic, i.e. for every transition $t \in T$, function $N(t, r)$ changes its values along the test process.

When we execute a test case on an implementation $I$ generated from a specification $S$, events in the enriched specification $S_E \in ETS(L)$ are updated with the number of executions in every recursion level. In this way, we maintain information concerning which behaviours or specification parts have not been sufficiently tested. Note that from the specifications described as ETS we can easily obtain risk and coverage values.

We assume that every transition has its own risk value. We also assume the existence of an heuristic function for measuring risks $f_{MR}(e, r, n) \rightarrow [0, R_I(e)]$ provided by the test engineer. This function will provide the risk measured for individual executions in a given level of recursion. This function must be convergent, and the sum over $r$ and $n$ of all risk measurements for a single event $e$ must be less than or equal to the risk of that event.
**Example 1** A suitable risk measurement function can be defined as

\[ MR_r^n(e) = \frac{R_I(e)}{2^{r+n}} \text{ for } r, n \geq 1 \]

Up to now, we have been considering transition systems without any additional information about which parts may be recursively called, which parts correspond to the main process, etc. In other words, when we traverse a plain LTS we do not know which states are recursively accessed from other states. With ETS, we consider every transition as a potential process (i.e. as a potential destination for a recursive call). Every time we reach a previously visited state, we assume that we have increased by one the recursive level for the next transition. In this way, we just need to check how many times we have visited a state to obtain the level of recursion.

![Diagram](image)

**Example 2** Suppose that we have the recursive specification \( S_E \in ETS(L) \) appearing in Figure 3. Function \( N(t,r) \) appears next to the corresponding label for every transition. We have represented the function \( N(t,r) \) as a sequence where the first element is the number of times we executed the transition in the first recursion level, the second element corresponds to the second level of recursion and so on. Initially, all values in the sequence are zero because we did not execute any test yet. Suppose also that we have a physical object \( I \) that implements correctly the behaviour described in the specification, i.e. \( I = S_E \), and that we want to execute test cases \( tc_1 \) and \( tc_2 \) described in Figure 3.

\( S_{bis} \) represents a snapshot of \( S_E \in ETS(L) \) after the execution of both test cases. Event \( a \) has been tested twice in the first level of recursion, one for each test case. Besides, this event has also been tested in the second level of recursion, which corresponds to the last transition of \( tc_1 \). The rest of the events have been executed only once in the initial recursion level.
3.2 Algorithms for Risk-driven Test Case Selection

For test generation and selection, we firstly adopted a classical testing algorithm and modified it to take into account risk and coverage values. The classical approach selected was Tretmans’ [14], which constructs tree-like deterministic test cases recursively selecting at random a subset of all possible specification transitions from a given state.

| Table 4
Generating test cases for $S$

| Given $S \in ETS(L)$, we construct a test case $tc := \sum\{a; tc_a | a \in A_{MR}\}$ recursively as follows:
| (i) Construct the set $C_S := \{Init(S') | S = \epsilon \Rightarrow S'\}$
| (ii) Among all possible sets $A \subseteq Init(S)$, select the set $A_{MR}$ having a maximum value of $\frac{\sum_{e \in A_{MR}}^\epsilon r, t(e)}{\text{Card}(A)}$ and satisfying one of the following:
  | (a) $\forall C \in C_S : A_{MR} \cap C \neq \emptyset$ and $v(tc) = \text{fail}$, or
  | (b) $\emptyset \in C_S$ and $A_{MR} = Init(S)$ and $v(tc) = \text{pass}$, or
  | (c) $A_{MR} = \emptyset$ and $v(tc) = \text{pass}$
| (iii) Construct recursively $tc_a$ as a test case for $\sum\{i; S' | S = a \Rightarrow S'\}$

(*) When representing a test case, $\sum$ represents branching and $a; s$ is short notation for transitions (i.e. $-a \rightarrow s$).

In our case (c.f. Table 4). We modified Tretmans algorithm to select (considering the conditions expressed in [14]) the set $A_{MR} \subseteq Init(s)$ that maximizes the mean risk measurement.

Concerning the test generation process and the ETS formalism, before we generate any test case, we make a copy of $S_E \in ETS(L)$ and name it $S_{bkp}^E$. During the generation process we will work with $S_{bkp}^E$ instead of $S_E$. Then, each time a new set $A_{MR}$ is selected, the values of $N(t, r)$ in copy $S_{bkp}^E$ are updated accordingly as they are executed. For example, if due to recursion the same transition is selected for a second time in the being generated test case, the corresponding value for $N(t, r)$ will reflect that now we are in the second level of recursion. These values are updated in $S_{bkp}^E$ and are considered a priori values (c.f section 2.4). In other words, a priori values are updated along the generation of a test case over the copy, and they guide the construction of the test case in a dynamic fashion.

Once a test case has been completely generated, we recover the original ETS specification, formerly $S_E$, and execute the test case. After the execution of the test case, values of $N(t, r)$ in $S_E$ are updated according to the execution sequence obtained a posteriori.

This cycle (i.e. test generation using a priori values, test execution to obtain a posteriori values, which are used as the initial values for the next iteration) is repeated until test cases with the desired coverage or cost are
obtained. This way, we construct dynamically test cases to cover those parts less adequately tested so far. This approach has been illustrated recently with a case study [4] and described extensively in [5].

Nevertheless, the algorithm in table 4 has two drawbacks:

(i) **Unnecessary cost increments**: the algorithm generates test cases with a tree structure introducing additional branches to cover non deterministic behaviours. When executing such test cases, they might examine certain parts of the implementation already tested, while others might not be covered enough, originating extra cost and decreasing effectiveness.

(ii) **Partial selection versus global selection**: the selection of $A_{MR}$, along the test case generation, has not considered any prediction level. This means that there could be cases where the chosen transitions have not been previously tested, but which drive to behaviours with a reduced impact over the global risk.

### Table 5
Generating test cases using prediction.

Given $S \in ETS(L)$, $i_p$, $l_{max}$ and $s_x = s_0$. A test case $tc$ of $S$ is:

$$tc := \{a; tc_a | \text{PathTr}(\varphi_{opt}) = a.a'\}$$

with $\varphi_{opt} \in \Gamma : \Gamma = \{\varphi \in \text{Path}(s_x) : |\varphi| \leq i_p\}$ that satisfy:

1. $MR_I(S, \varphi_{opt}) \geq MR_I(S, \varphi), \forall \varphi \in \Gamma$.
2. $|tc| \leq l_{max}$
3. Using $\text{PathTr}(tc) = \sigma.a$ we assign verdicts with:
   a) if $L \in \text{Ref}(S, \sigma)$ then $v(tc) = \text{pass}$;
   b) if $\{a\} \in \text{Ref}(S, \sigma)$ then $v(tc) = \text{inc}$;
   c) if $\{a\} \notin \text{Ref}(S, \sigma)$ then $v(tc) = \text{fail}$;

being $MR_I(S, \varphi) = MR_{ini}(S, \varphi) + \frac{MR_{end}(S, \varphi)}{1+\alpha.N_{inc}}$ and divided in $\varphi = \varphi_{ini} \cdot \varphi_{inc}$ where $\varphi_{ini}$ is the initial subpath $\varphi$ without inc verdicts and $\varphi_{end}$ is the ending subpath from the first inc verdict. $\alpha \in [0, 1]$ is a parameter we may select and $N_{inc}$ is the number of verdicts inc that have appeared. We calculate:

$$MR_{ini}(S, \varphi) = \sum_{t \in \varphi_{ini}} R^r_{t}(t)$$
$$MR_{end}(S, \varphi) = \sum_{t \in \varphi_{end}} R^r_{t}(t)$$

$tc_a$ is the test case generated from the state $s_y$ such that $s_x - a \rightarrow s_y$. Therefore, we want to complement the possibility of generating test cases
with a tree structure, e.g., the ones appearing in table 4, with the generation of test cases oriented to check certain behaviours poorly tested so far. The algorithm presented in table 5 can be used in the later test phases when some specification parts still have a low level of coverage. In such table, function \texttt{PathTr(})(\varphi)\texttt{)} returns the trace \(\sigma \in \text{Tr}(P)\), obtained following path \(\varphi\). Again, during test case generation we must use a copy \((S_{E}^{dikp})\) to modify its a priori \(N(t, r)\) values. The main properties of this new algorithm are:

(i) We introduce a \textit{prediction parameter} \((i_p)\) and a \textit{maximum length} \((l_{\text{max}})\).

(ii) From state \(s' \in \text{Stat}(S)\) we evaluate the risk of all possible transition paths \(\varphi \in \text{Path}(s')\) such that \(|\varphi| \leq i_p\), i.e., paths with less length than the prediction parameter.

(iii) We choose the path \(\varphi_{opt}\) that, a priori, measures more risk. Concerning risk measurement, we take care of the presence of verdicts \texttt{inc} using the parameter \(\alpha \in [0, 1]\). This parameter allows to penalize test cases that may end without a conclusive verdict, but generating a cost. If \(\alpha = 0\) then the presence of inconclusive verdicts is not considered. If \(\alpha = 1\), we reduce the \textit{a priori risk measurement}, computing the risk contained in the nondeterministic sequence and dividing its value by\((1 + N_{\text{inc}})\). A typical initial value may be \(\alpha = 0.5\).

(iv) Once \(\varphi_{opt}\) has been chosen, we take its first transition \(t\) and update the value of \(N(t, r)\) in \(S\) to model its execution, advance to the next state and repeat step 2 until the test case \(tc\) reaches the desired length.

Changing the prediction parameter \(i_p\) we may tune the precision when generating the test case. With \(i_p = 1\) we have the same information than in the algorithm presented in table 4. With \(i_p = \infty\) we will choose the \textit{(a priori) best test case}. The price we have to pay when increasing the value of \(i_p\) is the computational cost needed to evaluate all possible paths from a given state and the inherent risk measurement computations. Our experience shows that \(i_p\) values around 3 to 5 are quite feasible and specification realistic.

\textbf{Example 3} Figure 4 shows the specification \(S\). The risk values estimated for its events are: \(R_a = 2\), \(R_b = 1\), \(R_c = 5\), \(R_d = 3\) y \(R_e = 1\). Considering there is no recursion, we select the next function to measure the risk:

\[ f_{MR}(e,n) = \frac{R_i(e)}{2^n} \]

Such function satisfies:

\[ R_f(e) = \sum_{n=1}^{\infty} R_i^{n}(e) = \sum_{n=1}^{\infty} f_{MR}(e,n) = \sum_{n=1}^{\infty} \frac{R_i(e)}{2^n} \]

Therefore, in every execution we measure part of the risk for an event, and the global risk we may measure is equal to the risk of failure for the event.

Using the algorithm that appears in table 4 we may select for the set \(A\) one of the sets \(\{a\}\), \(\{d\}\) or \(\{a,d\}\). Their respective values for risk measurement are:
• $f_{MR}(a, 1) = 2/2 = 1$
• $f_{MR}(d, 1) = 3/2 = 1.5$
• $\frac{f_{MR}(a, 1) + f_{MR}(d, 1)}{\text{Card}([a, d])} = \frac{1+1.5}{2} = 1.25$

Hence, using this algorithm, we would choose $A = \{d\}$. Following the steps described in table 4, we obtain the test case $tc_1$, which appears in figure 4.

On the other hand, we will use the predictive algorithm of table 5 with a prediction parameter $i_p = 2$. In the initial state of $S_{bkp}^E$ we must calculate all transition sequences of length $i_p = 2$ and determinate their risk measurement. There are three cases:

(i) $a; b$: with the risk measurement $f_{MR}(a, 1) + f_{MR}(b, 1) = 2/2 + 1/2 = 1.5$
(ii) $a; c$: with the risk measurement $f_{MR}(a, 1) + f_{MR}(c, 1) = 2/2 + 5/2 = 3.5$
(iii) $d; e$: with the risk measurement $f_{MR}(d, 1) + f_{MR}(e, 1) = 3/2 + 1/2 = 2$

As the bigger measurement of risk is present in the second option, we take its first transition, modify a priori the values of $N(t, r)$ in $S_{bkp}^E$ for that transition, advance to the next state and repeat the procedure.

After the first transition there are only two possibilities $b$ or $c$, both of length 1. We proceed to determinate their risk measurement, which are: $f_{MR}(b, 1) = 1/2 = 0.5$ and $f_{MR}(c, 1) = 5/2 = 2.5$. Therefore, we choose the transition with $b$ obtaining the test case $tc_2$ in figure 4.

The a priori global risk measurement for $tc_1$ is $MR_I(S, tc_1) = 2$ and for $tc_2$ is $MR_I(S, tc_2) = 3.5$. The second test case is clearly better than the first concerning risk measurement. Thus, if the election of transitions is done with a certain level of prediction we can take advice of the information that an enriched transition system offers.

4 Conclusions

We have presented in this paper an approach to testing supported by formal methods, which also includes non-formal heuristics to introduce the experience of the testing engineer to evaluate the costs of the testing process.

Our experience showed us that this approach, based on error weighting and cost values, provides a way to assign values to different test cases, which
permits to classify them according to different criteria, taking into account the desired coverage and supported cost. Test generation can be directed by these heuristics to obtain context-adapted test suites.

This proposal has been experimented recently with a practical case study: the testing of a protocol for mobile auctions in a distributed, wireless environment [4]. LOTOS was selected as the supporting formal language. Nevertheless, the ideas discussed here are not specific to LOTOS, but applicable to a wide range of formal techniques, with comparable expressive power.

References


