Test Case Backward Generation for Communicating Systems from Event Logs

Sébastien Salva and Jarod Sue
LIMOS - UMR CNRS 6158, Clermont Auvergne University, UCA, France

Keywords: Communicating Systems, Test Case Generation, Test Case Mutation.

Abstract: This paper is concerned with generating test cases for communicating systems. Instead of considering that a complete and up-to-date specification is provided, we assume having an event log collected from an implementation. Event logs are indeed more and more considered for helping IT personnel understand and monitor system behaviours or performance. We propose an approach allowing to extract sessions and business knowledge from an event log and to generate an initial set of test cases under the form of abstract test models. The test architecture is adaptable and taken into consideration during this generation. Then, this approach applies 11 test case mutation operators on test cases to mimic possible failures. These operators, which are specialised to communicating systems, perform slight modifications by affecting the event sequences, the data, or injecting unexpected events. Executable test cases are finally derived from the test models.

1 INTRODUCTION

This paper focuses on the test of communicating systems, i.e. systems made up of concurrent components, e.g., Web services or Internet of things (IoT) components, which interact through a message passing protocol, e.g. HTTP. Testing such systems is known to be a hard process due to the problems inherent in controlling or monitoring many concurrent components interacting with one another simultaneously. Several works proposed solutions to develop test architectures, which include points of observation (PO), to collect information from the implementation and points of control and observation (PCO), to collect information but also to control the interactions with the implementation. Others proposed to apply test strategies, or to perform Model-based Testing (MbT), e.g., (Ulrich and König, 1999; van der Bijl et al., 2004; Cao et al., 2009; Torens and Ebrecht, 2010; Kanso et al., 2010; Aouadi et al., 2015; Hierons, 2001). Despite these significant contributions, generating test cases for this kind of systems is still difficult and long. With most of the approaches proposed in the literature, a complex specification must be written and verified, then a model-based testing approach is applied to generate test cases with respect to a static test architecture.

In the Industry, models are often neglected though. Writing models is indeed known to be a hard and error-prone task. Besides, keeping them up-to-date is also difficult especially over the long term. Instead of using specifications, we propose to rely on event logs, collected from a system under test, which we denote SUT. Although no specification is given, some approaches might be yet considered to generate test cases:

- a basic record and replay technique: from an event log, it is possible to extract sessions, i.e. sequences of correlated events interchanged among different components. Then, these sessions could be converted to executable test cases to evaluate SUT again. Unfortunately, this technique does not work well for communicating systems, as SUT may include non deterministic components, or not testable ones (not observable, or not controllable). Test cases must be adapted w.r.t. these properties;
- model learning followed by test case generation: model learning is a research field gathering algorithms specialised in the construction of models by inference. On the one hand, active model learning algorithms interact with SUT by means of tests to get sequences of reactions encoded with models. Few approaches are specialised to communicating systems, e.g., (Petrenko and Avelenda, 2019) and these ones are time-consuming and cannot produce large models. On the other hand, passive model learning approaches, e.g.,
(Mariani and Pastore, 2008; Beschastnikh et al., 2014; Salva and Blot, 2020) are able to generalise behaviours retrieved in large event logs and to encode them into models. But, this generalisation might lead to the generation of incorrect test cases leading to false positives. Once models are retrieved, classical MbT approaches can be applied to produce test cases. These approaches (re-)generate test data along with concrete execution paths to construct test cases. Again this is a time consuming activity.

This paper presents another approach to generating test cases for communicating systems from event logs, which is devised with the previous attention points in mind. The resulting test cases aim at experimenting the whole system. In short, our approach performs an analysis of event logs to extract some business knowledge, reverse-engineers abstract test models (but no specification) from event logs, mutates them with specialised mutation operators and finally generates executable test cases. The test architecture, i.e. the number of PCOs, POs along with their capabilities (what they are able to observe), is adaptable and taken into consideration while the test case generation. Test case mutation (Köröglo and 0001, 2018; Paiva et al., 2020) is an approach that builds new test cases by performing slight modifications on existing test cases in order to later check whether the system under test is resilient to errors, to security attacks etc. In other terms, we gather some advantages of passive model learning (extraction of sessions from event logs, knowledge extraction), which we associate with test case generation (test selection, generation of test cases w.r.t. the PCOs and POs) and test case mutation.

The paper is organised as follows: we provide some definitions used throughout the paper in Section 2. Our approach is presented in Section 3 with a motivating example. We present two algorithms to build test cases from event logs and 11 test case mutation operators specialised for communicating systems. Section 4 summarises our contributions and draws some perspectives for future work.

2 DEFINITIONS

We denote $E$ the set of events of the form $e(\alpha)$ with $e$ a label and $\alpha$ an assignment of parameters in $P$ to a value in the set of values $V$. We write $x := *$ the assignment of the parameter $x$ with an arbitrary element of $V$, which is not of interest. The concatenation of two event sequences $\sigma_1, \sigma_2 \in E^*$ is denoted $\sigma_1 \sigma_2$. $E^*$ denotes the empty sequence. For sake of readability, we also write $\sigma_1 \in \sigma_2$ when $\sigma_1$ is a (ordered) subsequence of the sequence $\sigma_2$. $\text{prefix}(\sigma)$ denotes the set of initial segments of $\sigma$.

We also use the following notations on events to make our algorithms more readable:

- $\text{from}(e(\alpha)) = c$ denotes the source of an event;
- $\text{to}(e(\alpha)) = c$ denotes the destination;
- $\text{isreq}(e(\alpha)), \text{isresp}(e(\alpha))$ are boolean expressions expressing the nature of the event.

We formulate a test case with a deterministic Input Output Labelled Transition System (IOLTS) having a tree form and whose sink states are either labelled by the test verdicts pass, fail, or inconclusive. Its transitions are labelled by events in $E \cup \{\emptyset\}$, with $\emptyset$ a special label expressing the absence of reaction (Phillips, 1987).

**Definition 1 (Test Case).** A test case is a deterministic IOLTS $\langle Q, q_0, \Sigma, \rightarrow \rangle$ where:

- $Q$ is a finite set of states; $Q$ contains three special states: pass, fail and inconclusive
- $q_0$ is the initial state;
- $(\Sigma \subseteq E) \cup \{\emptyset\}$ is the finite set of events. $\Sigma_f \subseteq \Sigma$ is the finite set of input events beginning with "?", $\Sigma_o \subseteq \Sigma$ is the finite set of output events beginning with "!", with $\Sigma_f \cap \Sigma_o = \emptyset$;
- $\rightarrow \subseteq Q \times \Sigma \cup \{\emptyset\} \times Q$ is a finite set of transitions

A transition $(q, e(\alpha), q')$ is also denoted $q \xrightarrow{e(\alpha)} q'$.

3 TEST CASE GENERATION AND MUTATION

The ability of our approach to generate test cases from an event log produced by a communicating system SUT, is dependent on the following realistic assumptions:

- **A1 Event Log**: the communications among the components can be monitored by POs using different techniques, e.g., wireless sniffers or server logs. These POs may have different observation capabilities. The latter are known and defined with a relation $PO : \text{Component} \times E \rightarrow \{\text{true, false}\}$. The resulting event logs are collected in a synchronous environment made up of synchronous communications. They include timestamps showing when the events occurred.
- **A2 Event Content**: components produce communication events or non-communication events. Both include parameter assignments allowing to identify the source and the destination of each
event. For non-communication events, both the source and the destination refer to the same component that has produced the event. Besides, a communication event can be identified either as a request or a response:

- **A3 Component Collaboration**: Workflows of events are correlated by means of parameter assignments.
- **A4 Component Controllability**: we assume knowing the set of components that can be experimented, which is denoted PCO.

\[ \forall c : \text{Component} \rightarrow \sum_{e \in \mathcal{E}} \text{PO}(c, e(\alpha)) = \text{true} \]

Figure 1: Example of system made up of 3 components; the test architecture has 3 POs and 1 PCO.

Let’s illustrate these notions of POs and their capabilities with the example of system and test architecture of Figure 1. Consider that we can observe for all the components \(c_1, c_2\) and \(c_3\) any received request and the associated response. For \(c_1\), this can be formulated by \(PO(c_1, e(\alpha)) : (\text{isreq}(e(\alpha)) \land to(e(\alpha)) = c) \lor (\text{isresp}(e(\alpha)) \land from(e(\alpha)) = c)\). Consider now that for \(c_2\) nothing can be observed (there is no point of observation), we have \(PO(c_2, e(\alpha)) : \text{false}\).

The relation \(PO\) may indirectly be used to evaluate the capability of observation for every component. For a component \(c\), we define this measure as follows: \(obs(c) = \sum_{e(\alpha) \in \mathcal{E}} PO(c, e(\alpha)) = \text{true}\). It is worth noting that dealing with all the potential events is not required; it is usually sufficient to consider the five event types illustrated in Figure 2.

Figure 2: Classical event types that can be considered for measuring PO observability.

The next step applies 11 mutation operators on the initial test case set, which modify events or data. These operators mostly produce test cases dedicated to test the robustness or security of SUT. Finally, every test case tree is converted into an executable test case. These steps are detailed below.

### 3.1 Session Extraction

The recovery of sessions from an event log is quite straightforward when the parameters used to identify sessions, namely a correlation set, is known in advance. When, this is not the case, we proposed in (Salva et al., 2021) an algorithm (and a tool) to retrieve sessions by exploring the correlation set space that can be derived from an event log. Our algorithm can return the best session set that meet a predefined set of quality attributes defined by the user, or returns several solutions and sort them from the best to the lowest quality.

The tool returns a set \(S\) of sessions along with a correlation set \(\text{Corr}(\sigma)\) for every session \(\sigma \in S\). All the assignments \(p := v\) found in \(\sigma\) whose parameters are also used in \(\text{Corr}(\sigma)\) are replaced by \(p := \ast\).

### 3.2 Session Clustering and Knowledge Extraction

This step takes as input the session set \(S\) and builds a set of abstract traces of the form \(<e_1(\alpha_1), l_1>\).
Def 2 (Abstract Traces). Let $L$ be a set of labels. An abstract trace is a sequence $\langle e_1(\alpha_1), l_1 \rangle \cdots \langle e_k(\alpha_k), l_k \rangle$ such that the parameter values are replaced by \("\ast\) except for the parameters from, to. $l_1, \ldots, l_k$ are label lists expressing some business knowledge about the sessions, e.g., \("\ast\) or \("\text{crash}\)."

To build abstract traces, we begin to partition the session set $S$ into classes of equivalent sessions. We formulate that two sessions are equivalent if they share the same sequence of abstract events. Given an event $e(\alpha)$, an abstract event $e(\alpha')$ simply results from the replacement of the parameter values by \("\ast\) excluding the parameters from and to. The equivalence relation between two sessions is defined by means of a projection, which performs this event abstraction:

Def 3. Two event sequences $\sigma_1, \sigma_2 \in \mathcal{E}^*$, are equivalent, denoted $\sigma_1 \sim \sigma_2$, if $\exists Q \in \{ \text{from}, \text{to} \}$ with: $\text{proj}_{ \{ \text{from}, \text{to} \} } \sigma_1 = \text{proj}_{ \{ Q \} } \sigma_2$, such that $\text{proj}^{Q} : \mathcal{E}^* \rightarrow \mathcal{E}^*$ is the projection $e_1(\alpha_1') \cdots e_k(\alpha_k') = \text{proj}^{Q}(e_1(\alpha_1) \cdots e_k(\alpha_k))$ and $\alpha'_i = \{ x := \ast \mid x \in \alpha_i \land x \notin Q \} \cup \{ x := \ast \mid x := v \in \alpha_i \land x \notin Q \}$.

Now, given a class of equivalent sessions $c = \{ \sigma_1, \ldots, \sigma_n \}$, we analyse the events and parameter values to extract knowledge by means of an expert system. The latter is an inference engine that applies a set of rules on a base of facts. In our context, the former base of facts is a session set. The rules encode expert knowledge about communicating systems and build abstract traces. It is worth noting that an expert system offers the benefit to save time by allowing its reuse on several communicating systems.

We represent inference rules with this format: When conditions on facts Then actions on facts (format taken by the Drools inference engine\(^1\)). To ensure that this step is performed in a finite time and in a deterministic way, the inference rules have to meet these hypotheses:

- Finite Complexity: a rule can only be applied a limited number of times on the same knowledge base,
- Soundness: the inference rules are Modus Ponens (simple implications that lead to sound facts if the original facts are true).

We devised inference rules that analyse event content or event sequences to recognise crashes, errors, the authentication of users, and the generation of Access tokens, which temporally provide accesses to specific user’s data. Figure 4 Figure 4 exemplifies a rule for recognizing a system crash with an HTTP status of above 500. It creates an abstract event having a new label \("\text{crash}\)."

Once the equivalent class $c = \{ \sigma_1, \ldots, \sigma_n \}$ has been analysed by the expert system, we obtain one abstract trace of the form $\langle e_1(\alpha'_1), l_1 \rangle \cdots \langle e_k(\alpha'_k), l_k \rangle$. From the equivalent classes of sessions $c_1, \ldots, c_m$, we obtain a set of $n$ abstract traces, which is denoted $ATraces$.

Figure 5 illustrates an example of 3 sessions extracted from an event log, itself produced by 3 components API, Products and Pay, using the architecture of Figure 1. The two first sessions are equivalent as they share the same sequence of labels and parameters with different data. Our approach builds two clusters of equivalent sessions $c_1(\{h_1\})$ and $c_2(\{h_2\})$. It also builds 2 abstract traces, $t_1$ includes one new label \("\text{crash}\) as the expert system has detected that the last event expresses a crash of the component Pay.

3.3 Test Case Generation

Test cases are generated in the form of IOLTS trees whose final states are labelled by a verdict. The use of the IOLTS formalism allows to synthesize generic test cases from which, written with various languages, can be derived concrete test scripts. Besides, it is easier to define transformations or mutations on IOLTS test cases.

Test cases are built using the set PCO and the relation PO while taking the controllability and observability of the components into account. At this point, a user may decide to adapt the system’s POs with regards to the available tooling. As the initial event log has been collected from every component $c$ in accordance with a certain level of observability given by $\text{obs}(c)$, the user may only lower this level by redefining $\text{PO}(c, e(\alpha))$.

The test case generation is implemented by Algorithms 1 and 2. Algorithm 1 takes as input a set of abstract traces $ATraces$ and returns the set $\text{SelectedATraces}$. As the set $ATraces$ may be large, it performs a selection according to 3 criteria (lines

\[ \text{rule } \ast \text{LabelCrash 1}\]
\[ \text{when}\]
\[ \text{\$ev: Event(paramStatus>500);}\]
\[ \text{then}\]
\[ \text{insert(new Aevent($\text{\$ev, L("crash")}$));}\]
\[ \text{end}\]

Figure 4: Inference rule example.

\[ \text{\footnote{https://www.drools.org/}} \]
1. Crash/Error Coverage: it selects in priority all the abstract traces whose at least one event expresses a crash or an error;

2. Component Coverage: it completes them with abstract traces in such a way that every component of SUT will be covered by the tests. This is performed with the relation $<_c$, which sorts the abstract traces having the most events produced by components not yet referenced in SelectedATraces: $<_c: t_1 <_c t_2$ if $\text{from}(t_1) \cup \text{from}(\text{SelectedATraces}) > \text{from}(t_2) \cup \text{from}(\text{SelectedATraces})$, with $\text{from}(E) = \bigcup_{e \in E} \{ e(\alpha) \mid e(\alpha), l \in \sigma \}$.

3. Event Coverage: it also selects other abstract traces until the ratio of selected events over the total number of events reaches a given threshold.

These events are selected with the relation $<_e$, which orders the abstract traces having the most events not used in SelectedATraces: $t_1 <_e t_2$ if $\text{event}(t_1) \cup \text{event}(\text{SelectedATraces}) > \text{event}(t_2) \cup \text{event}(\text{SelectedATraces})$, with $\text{event}(E) = \bigcup_{e \in E} \{ e(\alpha) \mid e(\alpha), l \in \sigma \}$.

Then, Algorithm 1 adapts the selected abstract traces with respect to PCO and PO. It only keeps the abstract traces whose first event is performed by a component $c$ that can be experimented ($c \in \text{PCO}$). Besides, the algorithm also calls the procedure InOut, which covers the events of an abstract trace $t$ to insert the notion of input and output. Meanwhile, the procedure filters out the unobservable events performed by a component with respect to the relation PO (the event $e(\alpha)$ such that $\text{PO}(\text{from}(e(\alpha)), e(\alpha)) \vee \text{PO}(\text{to}(e(\alpha)), e(\alpha))$ is false is not kept to build a test case line(17)).

Algorithm 2 takes the set SelectedATraces and produces IOLTS test cases. Given an abstract trace $t$, the algorithm selects one event sequence of the class $c(t)$ and builds an initial test case $tc$ composed of parameter values (lines 2-4). The IOLTS $tc$ is derived by means of the operator $\text{Its} : (E \times
L)^* × E^* × \{fail, pass\} → IOLTS, which takes an abstract trace, its related event sequence, a verdict \(v\) and returns IOLTS \((Q, σ, Σ, →)\) defined by the rule \(<e_1(α_1), l_1 > < e_1(α_2), l_2 > < e_1(α_3) \ldots e_k(α_k), v → q_0 \sigma_{e_1(α_1)}l_1 → q_1 \ldots q_{k-1} \sigma_{e_k(α_k)} l_k → v\). A verdict \(v\) of \(tc\) is established by means of the labels found in the abstract trace. This test verdict denoted \(v < e_1(α_1), l_1 > < e_1(α_k), l_k \geq 1\) is fail iff \(\exists i \leq k: "crash" ∈ l_i\), otherwise \(v < e_1(α_1), l_1 > < e_1(α_k), l_k \geq 1\) is pass.

Algorithm 2 now checks whether other events may be observed after experimenting SUT with an arbitrary event sequence of \(prefix(traces(tc))\) (lines 5-9). After an execution of events in \(prefix(traces(tc))\), depending on the SUT internal states, one might indeed observe the same output event with different parameter assignments, or different output events, or the fact that SUT does not react (formulated with the label \(θ\)). Given an event sequence \(σ\) (collected from SUT), we formulate this notion of observation after the execution of \(σp ∈ \sigma\):

\[
out(σ, σ_p) = \left\{ \begin{array}{ll}
\{e(σ)\} & \text{if } σ_p \cap e(σ) \in σ \\
\emptyset & \text{otherwise}
\end{array} \right.
\]

We can now say that SUT may produce two different observations after being experimented with the event sequence \(σp\) iff it exists two sessions \(σ1\) and \(σ2\) such that \(σp \in prefix(σ1) \cap prefix(σ2)\) and \(out(σ1, σ_p) \neq out(σ2, σ_p)\). Algorithm 2 hence covers every abstract trace \(t\) and session \(σp\) in \(cl(t)\) to check whether different observations are possible against the current test case \(tc\) (line 5). If this happens, an IOLTS \(tc_2\) is built from \(σ2\) and \(tc\) is completed with \(tc_2\) by means of the parallel synchronisation operator \(∥\).

Furthermore, as event logs do not necessarily encode all the behaviours of SUT, the test case \(tc\) is completed (line 10) with the operator \(compl : IOLTS → IOLTS\) defined by these rules:

\[
r_1 : q_1 \xrightarrow{γ_e(α)} q_2, q_2 \in \{pass, fail\} \vdash q_1 \xrightarrow{γ_e(α)} q_11 \rightarrow q_2, q_11 \rightarrow fail, q_1 \rightarrow fail\\
r_2 : q_1 \xrightarrow{γ_e(α)} q_2 \notin \{pass, fail\} \vdash q_1 \xrightarrow{γ_e(α)} q_2, q_1 \xrightarrow{γ_e(α)} fail\\
r_3 : q_1 \xrightarrow{γ_e(α)} q_2 \rightarrow q_1 \xrightarrow{γ_e(α)} q_1 \xrightarrow{γ_e(α)} q_2, q_2 \rightarrow in\text{conclusive}\\
r_4 : q_1 \xrightarrow{γ_e(α)} q_2, q_1 \xrightarrow{γ_e(α)} q_3 \notin ∩ q_1 \rightarrow fail
\]

The inference rule \(r_1\) means that when the test case \(tc\) is finished by an input event, a transition to a verdict state and labelled with \(θ\) is added to formulate that the absence of event is expected. Two transitions to fail are added to express that the observation of any other output event (label \(!r\)) is incorrect. Similarly, \(r_2\) targets the other transitions labelled by input events. \(r_3\) completes the test case when an output event is expected. It adds a new transition \(q_1 \rightarrow inconclusive\) modelling that we cannot conclude whether the behaviour is correct when we observe any other unexpected output event from \(q_1\). \(r_4\) completes the previous rule in the case there are only outgoing transitions labelled by output events from \(q_1\). The rule adds a transition to fail modelling that the observation of no reaction is faulty.

Algorithm 2

```
input : ATraces
output : TC
1 foreach \(t = < e_1(α_1), l_1 > < e_2(α_2), l_2 > ∈ ATraces\) do
2 Choose arbitrary \(σ ∈ cl(t)\) :
3 \(cl(t) = cl(t) \setminus \{σ\}\);
4 \(tc = lst(σ,v(\theta))\);
5 foreach \(σp ∈ ATraces, σp ∈ cl(tc)\) such that
6 \(σp ∈ \bigcup_{l ∈ \text{labels}} T_r\) \(\cap \text{prefix}(σp)\) \(\cap \text{prefix}(σp)\): \(out(σ, σ_p) \neq out(σ_2, σ_p)\) do
7 \(t_{c2} = lst(σp, v(\theta));\)
8 \(cl(t_{c2}) = cl(t_{c2}) \cup \{σp\};\)
9 \(tc = tc ∥ t_{c2};\)
10 \(ATraces = ATraces \setminus \{tc\};\)
11 \(TC = TC ∪ \{tc\};\)
12 \(ATraces = ATraces \setminus \{tc\};\)
```

Figure 6: Test case example.

Figure 6 illustrates a test case obtained from the abstract traces of Figure 5 after having removed the PO for the component "Products" \((PO("Products", e(α)) : false)\). The events /supply() and the response ok() were hence removed from the test case to comply with the new test architecture.
### Table 1: Mutation operators and short description.

<table>
<thead>
<tr>
<th>Name</th>
<th>Def.</th>
<th>Desc.: all the test cases that can be derived from tc by</th>
<th>Expected</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event Duplication</td>
<td>$\forall e(\alpha), L \in \Sigma_1, add(inc(\forall e(\alpha), L), \forall e(\alpha), L))$</td>
<td>adding just before $\forall e(\alpha)$ a copy of the event followed by its related response</td>
<td>at least the response, no error, no crash</td>
</tr>
<tr>
<td>Event Swapping</td>
<td>$\forall \forall e(\alpha), L \in \Sigma_2, swap(inc(\forall e(\alpha), L), \forall e(\alpha), L))$</td>
<td>swapping two different input events and their related responses along with all their nested output events</td>
<td>no crash</td>
</tr>
<tr>
<td>Event Removal</td>
<td>$\forall \forall e(\alpha), L \in \Sigma_1, del(tc, \forall e(\alpha), L))$</td>
<td>deleting an input event and its related response along with all the nested output events. tc must still hold at least one input event</td>
<td>no crash</td>
</tr>
<tr>
<td>HTTP Verb Change</td>
<td>$\forall e(\alpha), L \in \Sigma_1, Verb(tc, \forall e(\alpha), L))$</td>
<td>changing the HTTP verb of the event if available</td>
<td>no crash</td>
</tr>
<tr>
<td>Data Alteration</td>
<td>$\forall e(\alpha), L \in \Sigma_1, dataAlt(\alpha) \forall f r o m = c_1, t o = c_2, session_id := id1, token := id2{e(\alpha), L}$</td>
<td>randomly changing the data</td>
<td>no crash</td>
</tr>
<tr>
<td>Delay Addition</td>
<td>$\forall e(\alpha), L \in \Sigma_1, add(tc, \forall e(\alpha), L, 0)$</td>
<td>adding a delay during which no reaction should be observed before every input event</td>
<td>no error, no crash</td>
</tr>
<tr>
<td>Token Removal</td>
<td>$\forall e(\alpha), L \in \Sigma_1, tokenAlt \in L : delToken(tc, \alpha)$</td>
<td>delete the token in the input event</td>
<td>error, no crash</td>
</tr>
<tr>
<td>Token Alteration</td>
<td>$\forall e(\alpha), L \in \Sigma_1, tokenAlt \in L : tokenAlt(tc, \alpha)$</td>
<td>replacing a token by another one</td>
<td>error, no crash</td>
</tr>
<tr>
<td>Session id Alteration</td>
<td>$\forall e(\alpha), L \in \Sigma_1, sessionIdAlt(tc, \alpha)$</td>
<td>replacing a session id by another one</td>
<td>error, no crash</td>
</tr>
<tr>
<td>Session Closure</td>
<td>$\forall e(\alpha), L \in \Sigma_1, closingSession(tc)$</td>
<td>adding a long delay and an input event at the end of every branch of the test case to check whether the session is terminated</td>
<td>error, no crash</td>
</tr>
<tr>
<td>Stress Testing</td>
<td>$\forall e(\alpha), L \in \Sigma_1, add(out(tc, \forall e(\alpha), L))$</td>
<td>adding just before $\forall e(\alpha)$ 100 copies of the event with overwhelming data</td>
<td>no error, no crash</td>
</tr>
</tbody>
</table>

#### 3.4 Test Case Mutation

We now have an initial set TC of test cases, which somehow mimic the behaviours encoded in the event log w.r.t. observability and controllability of the components. This test case set is completed to experiment SUT with further executions. To generate additional test cases, we apply 11 mutation operators on TC whose purposes are to perform slight test case modifications to mimic possible failures. These modifications may affect the sequence of events (Event duplication, swapping, removal), change data (HTTP Verb change, Data alteration, Token removal, Alteration, Session id Alteration) or add unexpected events (Delay Addition, Session Closure, Stress Testing).

The list of mutation operators along with short descriptions are given in Table 1. Several operators refer to the notion of nested events, which occur between a request $q_1 \xrightarrow{\forall req(\alpha)} q_2$ and its response $q_1 \xrightarrow{\forall resp(\alpha'), \forall f} q_2$. These nested events have to be taken into account while the test case mutation. For instance, for Event Removal, if the operator deletes $\forall req(\alpha)$ and its response into a test case, it also needs to delete the potential nested requests and responses $\forall Req(\forall e(\alpha_k))$ such that to $\forall Req(\alpha) = from(\forall Req(\alpha_1), to(\forall Resp(\alpha_1)) = from(\forall Resp(\alpha_{i+1})(\alpha_{i+1})|1 \leq i < k$ and to $\forall Resp(\alpha_k)) = from(\forall Resp(\alpha_{i}))$. 

![Figure 7: Test case example obtained with the mutation operator “HTTP Verb Change”](image)

If we take back the example of Figure 6 and if we apply the operator “HTTP Verb Change” we obtain the test case of Figure 7. The verb Post was replaced by Delete in the request/order; any output event not expressing a crash is allowed.

Finally, executable test cases are generated from the IOLTS trees. Some input events may still have assignments of the form $p:=\ast$. These ones refer to parameter that belong to a correlation set. The first input event of a test case tc is completed by means of the data found in Corr($\sigma$) and translated into source code. Every other input event of tc is also translated but the source code refers here to the correlation sets recovered in the output events preceding this input.
4 CONCLUSION

We proposed in this paper a solution to generate test cases for communicating systems from event logs. Instead of proposing a basic "record and replay" technique or an approach combining model learning and MBT, we presented algorithms allowing to extract knowledge by means of an expert system and generate an initial test suite made up of IOLTS test cases. By doing this, we intend to extract test verdicts, save computation time, and avoid the imprecision brought by models produced with passive model learning techniques. Besides, we proposed 11 test case mutation operators to expand the initial test case set.

We have implemented this approach in a tool prototype. Due to lack of room, we briefly summarise its features here: the tool is specialised for Web service compositions. It takes event logs as inputs and generates sessions by means of the tool presented in (Salva et al., 2021). The tool Drools is the expert system used to analyse sessions. Then, our tools produce test cases written with the Citrus framework. A complete evaluation will be presented in a future work.

REFERENCES


2https://citrusframework.org