Test Case Backward Generation for Communicating Systems from Event Logs

Sébastien Salva[1] and Jarod Sue[1]
[1]*LIMOS - UMR CNRS 6158, Clermont Auvergne University, UCA, France*
sebastien.salva@uca.fr, jarod.sue@uca.fr

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 Avellaneda, 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 (Köroglu 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 $\mathcal{E}$ the set of events of the form $e(\alpha)$ with e a label and α 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 $\sigma_1, \sigma_2 \in \mathcal{E}^*$ is denoted $\sigma_1.\sigma_2$. ε denotes the empty sequence. For sake of readability, we also write $\sigma_1 \in \sigma_2$ when σ_1 is a (ordered) subsequence of the sequence σ_2. $prefix(\sigma)$ denotes the set of initial segments of σ.

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

- $from(e(\alpha)) = c$ denotes the source of an event;
- $to(e(\alpha)) = c$ denotes the destination;
- $isreq(e(\alpha))$, $isresp(e(\alpha))$ 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 $\mathcal{E} \cup \{\theta\}$, with θ a special label expressing the absence of reaction (Phillips, 1987).

Definition 1 (Test Case) *A test case is a deterministic IOLTS $\langle Q, q0, \Sigma, \rightarrow \rangle$ where:*

- *Q is a finite set of states; Q contains three special states: pass, fail and inconclusive*
- *$q0$ is the initial state;*
- *$(\Sigma \subseteq \mathcal{E}) \cup \{\theta\}$ is the finite set of events. $\Sigma_I \subseteq \Sigma$ is the finite set of input events beginning with "?", $\Sigma_O \subseteq \Sigma$ is the finite set of output events beginning with "!", with $\Sigma_O \cap \Sigma_I = \emptyset$;*
- *$\rightarrow \subseteq Q \times \Sigma \cup \{\theta\} \times Q$ 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 $PO : Component \times \mathcal{E} \rightarrow \{true, false\}$. 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*;

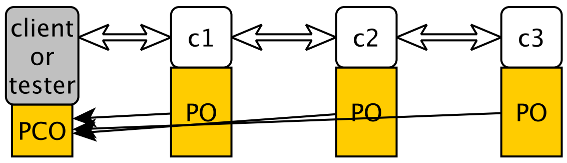


Figure 1: Example of system made up of 3 components; the test architecture has 3 POs and 1 PCO

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 $PO(c_1, e(\alpha)) : (isreq(e(\alpha)) \wedge to(e(\alpha)) == c) \vee (isresp(e(\alpha)) \wedge from(e(\alpha)) == c)$. Consider now that for c_2 nothing can be observed (there is no point of observation), we have $PO(c_2, e(\alpha)) : false$.

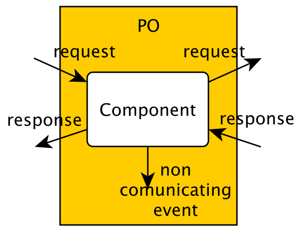


Figure 2: Classical event types that can be considered for measuring PO observability

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: $obs(c) = \sum_{e(\alpha) \in \mathcal{E}} PO(c, e(\alpha)) == true$. 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 3 illustrates the successive steps of the test case generation. The event log is initially segmented into sessions by means of A3. As a lot of sessions may express the same behaviours of SUT, Step 2 assembles those expressing similar behaviours and covers them to extract knowledge, e.g., the fact that an event represents an error or a failure, the fact that an event sequence represents a token generation, etc. This knowledge, expressed under the form of labels will be used for the test case generation and mutation. This step returns a set of sequences of elements $< e(\alpha), l >$, with $e(\alpha)$ an event whose parameter values are hidden and l a list of labels expressing knowledge. These sequences are called abstract traces.

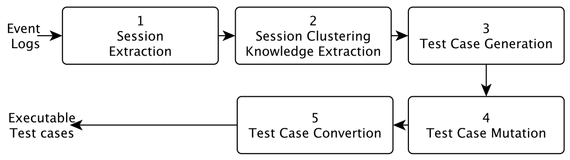


Figure 3: Approach Overview

Step 3 generates a first test case set from abstract traces. It applies a selection of abstract traces to keep in priority those that should trigger crashes and errors, and those allowing to cover the most components and events. This step also transforms the selected abstract traces to insert the notion of input and output and to take into consideration the capabilities of the POs. Then, test cases, given under the form of IOLTS trees, are generated.

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 $Corr(\sigma)$ for every session $\sigma \in S$. All the assignments $p := v$ found in σ whose parameters are also used in $Corr(\sigma)$ are replaced by $p := *$.

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 $< e_1(\alpha_1), l_1 > \cdots < e_k(\alpha_k), l_k >$ such that the parameter values are replaced by "*" except for the parameters from, to. $l_1, \ldots, l_k$ are label lists expressing some business knowledge about the sessions, e.g., "error" or "crash".

Definition 2 (Abstract Traces) *Let L be a set of labels. An abstract trace is a sequence* $< e_1(\alpha_1), l_1 > \cdots < e_k(\alpha_k), l_k > \in (\mathcal{E} \times L)^*$ *such that* $e_i(\alpha_i)_{1 \leq i \leq k} \in \mathcal{E}$*, and every parameter in* $P \setminus \{from, to\}$ *is assigned to "*" and* $l_i \subseteq L (1 \leq i \leq k)$ *is a set of labels.*

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 "*" excluding the parameters from and to. The equivalence relation between two sessions is defined by means of a projection, which performs this event abstraction:

Definition 3 *Two event sequences* σ_1 $\sigma_2 \in \mathcal{E}^*$*, are equivalent, denoted* $\sigma_1 \sim_b \sigma_2$ *iff* $proj_{\{from,to\}}\sigma_1 = proj_{\{from,to\}}\sigma_2$ *with:* $proj_Q : \mathcal{E}^* \rightarrow \mathcal{E}^*$ *is the projection* $e_1(\alpha'_1) \ldots e_k(\alpha'_k) = proj_Q(e_1(\alpha_1) \ldots e_k(\alpha_k))$ *and* $\alpha'_i = \{x := * \mid x := v \in \alpha_i \wedge x \notin Q\} \cup \{x := v \mid x := v \in \alpha_i \wedge x \in Q\}$

Now, given a class of equivalent sessions $cl = \{\sigma_1, \ldots, \sigma_m\}$, 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,

[1]https://www.drools.org/

```
rule "LabelCrash 1"
when
$ev: Event(paramStatus>=500);
then
insert(new Aevent($ev, L("crash"));
end
```

Figure 4: Inference rule example

- 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 "crash".

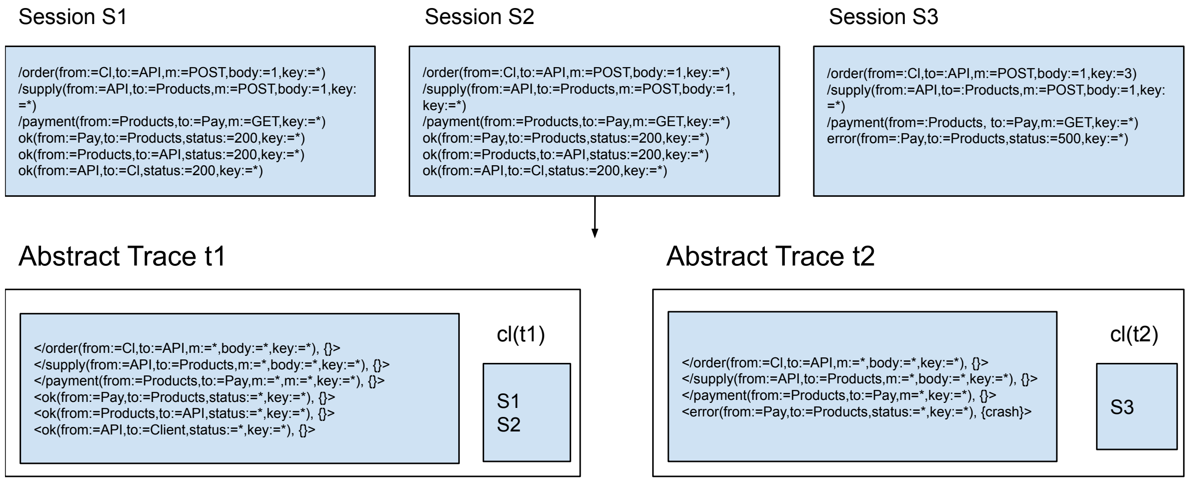
Once the equivalent class $cl = \{\sigma_1, \ldots, \sigma_m\}$ has been analysed by the expert system, we obtain one abstract trace of the form $< e_1(\alpha'_1), l_1 > \cdots < e_k(\alpha'_k), l_k >$. From the equivalent classes of sessions $cl_1, \ldots, cl_n$, we obtain a set of n abstract traces, which is denoted $ATraces$.

Figure 5 illustrates an example of 3 sessions extracted from an event log, itself produced by 3 components API, Produts 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 $cl(t_1)$ and $cl(t_2)$. It also builds 2 abstract traces. t_2 includes one new label "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 $obc(c)$, the user may only lower this level by redefining $PO(c, e(\alpha))$.



```
/order(from:=Cl,to:=API,m:=POST,body:=1,key:=*)
/supply(from:=API,to:=Products,m:=POST,body:=1,key:
=*)
/payment(from:=Products,to:=Pay,m:=GET,key:=*)
ok(from:=Pay,to:=Products,status:=200,key:=*)
ok(from:=Products,to:=API,status:=200,key:=*)
ok(from:=API,to:=Cl,status:=200,key:=*)
```

```
/order(from=:Cl,to:=API,m:=POST,body:=1,key:=*)
/supply(from:=API,to:=Products,m:=POST,body:=1,
key:=*)
/payment(from:=Products,to:=Pay,m:=GET,key:=*)
ok(from:=Pay,to:=Products,status:=200,key:=*)
ok(from:=Products,to:=API,status:=200,key:=*)
ok(from:=API,to:=Cl,status:=200,key:=*)
```

```
/order(from=:Cl,to=:API,m:=POST,body:=1,key:=3)
/supply(from:=API,to=:Products,m:=POST,body:=1,key:
=*)
/payment(from=:Products, to:=Pay,m:=GET,key:=*)
error(from=:Pay,to:=Products,status:=500,key:=*)
```

```
</order(from:=Cl,to:=API,m:=*,body:=*,key:=*), {}>
</supply(from:=API,to:=Products,m:=*,body:=*,key:=*), {}>
</payment(from:=Products,to:=Pay,m:=*,m:=*,key:=*), {}>
<ok(from:=Pay,to:=Products,status:=*,key:=*), {}>
<ok(from:=Products,to:=API,status:=*,key:=*), {}>
<ok(from:=API,to:=Client,status:=*,key:=*), {}>
```

```
</order(from:=Cl,to:=API,m:=*,body:=*,key:=*), {}>
</supply(from:=API,to:=Products,m:=*,body:=*,key:=*), {}>
</payment(from:=Products,to:=Pay,m=*,key:=*), {}>
<error(from:=Pay,to:=Products,status:=*,key:=*), {crash}>
```

Figure 5: Example of sessions and abstract traces

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 $SelectedATraces$. As the set $ATraces$ may be large, it performs a selection according to 3 criteria (lines 1-11):

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 $SelectedATraces$: $<_c$: $t_1 <_c t_2$ iff $|from(\{t_1\}) \setminus from(SelectedATraces)| >= |from(\{t_2\}) \setminus from(SelectedATraces)|$, with $from(E) = \bigcup_{\sigma \in E}\{from(e(\alpha)) \mid < e(\alpha), l > \in \sigma\}$;
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 $SelectedATraces$: $t_1 <_e t_2$ iff $|event(\{t_1\}) \setminus event(SelectedATraces)| >= |event(\{t_2\}) \setminus event(SelectedATraces)|$ with $event(E) = \bigcup_{\sigma \in E}\{e(\alpha) \mid < e(\alpha), l > \in \sigma\}$.

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 ($c \in PCO$). 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 $PO(from(e_i(\alpha_i)), e_i(\alpha_i)) \vee PO(to(e_i(\alpha_i), e_i(\alpha_i))$ is false is not kept to build a test case line(17)).

Algorithm 1:

```
   input  : $ATraces$
   output: $SelectedATraces$
1  foreach $t \in ATraces$ such that
     $\exists 1 \leq i \leq k : "error" \in l_i \vee "crash" \in l_i$ do
2      if $to(e_1(\alpha_1)) \in PCO$ then
3          $InOut(t)$;
4  while $from(SelectedATraces) \neq from(Atraces)$ do
5      Order traces in $ATraces$ w.r.t. $<_c$ and take the first trace $t$;
6      if $to(e_1(\alpha_1)) \in PCO$ then
7          $InOut(t)$;
8  while $Aevent(SelectedATraces)/Aevent(Atraces) < treshold$ do
9      Order traces in $ATraces$ w.r.t. $<_t$ and take the first trace $t$;
10     if $to(e_1(\alpha_1)) \in PCO$ then
11         $InOut(t)$;
12 Procedure $InOut( t =< e_1(\alpha_1), l_1 > \cdots < e_k(\alpha_k), l_k > )$: is
13     $t_2 := \varepsilon; cl(t_2) := cl(t)$;
14     for $1 \leq i \leq k$ do
15         if $(isreq(e_i(\alpha_i)) \wedge (to(e_i(\alpha_i) ==$
             $to(e_1(\alpha_1)) \wedge (from(e_i(\alpha_i) == from(e_1(\alpha_1))$ then
16             $t_2 := t_2. <?e_i(\alpha_i), l_i >$;
17         if $PO(from(e_i(\alpha_i)), e_i(\alpha_i)) \vee PO(to(e_i(\alpha_i)), e_i(\alpha_i))$
             then
18             $t_2 := t_2. <!e_i(\alpha_i), l_i >$;
19         if $(isresp(e_i(\alpha_i)) \wedge (from(e_i(\alpha_i) == c)$ then
20             $t_2 := t_2. <!e_i(\alpha_i), l_i >$;
21     Update $cl(t_2)$ w.r.t. $t_2$;
22     $SelectedATraces := SelectedATraces \cup \{t_2\}$;
23     $ATraces := ATraces \setminus \{t\}$;
```

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 $cl(t)$ and builds an initial test case tc composed of parameter values (lines 2-4). The IOLTS tc is derived by means of the operator $lts : (\mathcal{E} \times L)^* \times \mathcal{E}^* \times \{fail, pass\} \rightarrow IOLTS$, which takes an abstract trace, its related event sequence, a verdict v and returns an IOLTS $\langle Q, q0, \Sigma, \rightarrow \rangle$ defined by the rule $< e_1(\alpha'_1), l_1 > \cdots < e_k(\alpha'_k), l_k >, e_1(\alpha_1) \ldots e_k(\alpha_k), v \vdash q_0 \xrightarrow{e_1(\alpha_1), l_1} q_1 \ldots q_{k-1} \xrightarrow{e_k(\alpha_k), l_k} v$. A verdict v of tc is established by means of the labels found in the abstract trace. This test verdict denoted $v(< e_1(\alpha_1), l_1 > \cdots < e_k(\alpha_k), l_k >$ is fail iff $\exists 1 \leq i \leq k :$ "$crash$" $\in l_i$, otherwise $v(< e_1(\alpha_1), l_1 > \cdots < e_k(\alpha_k), l_k >$ 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 wit the label θ). Given an event sequence σ (collected from SUT), we formulate this notion of observation after the execution of $\sigma_p \in \sigma$ with:

$$out(\sigma, \sigma_p) = \begin{cases} \{!e(\alpha)\} & \text{if } \sigma_p !e(\alpha) \in \sigma \\ \{\theta\} & \text{otherwise} \end{cases}$$

We can now say that SUT may produce two different observations after being experimented with the event sequence σ_p iff it exists two sessions σ_1 and σ_2 such that $\sigma_p \in prefix(\sigma_1) \cap prefix(\sigma_2)$ and $out(\sigma_1, \sigma_p) \neq out(\sigma_2, \sigma_p)$. Algorithm 2 hence covers every abstract trace t_2 and session σ_2 in $cl(t_2)$ 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 σ_2 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 $compl : IOLTS \rightarrow IOLTS$ defined by these rules:

$$r_1 : q_1 \xrightarrow{?e(\alpha), l} q_2, q_2 \in \{pass, fail\} \vdash q_1 \xrightarrow{?e(\alpha), l} q_{11} \xrightarrow{\theta} q_2, q_{11} \xrightarrow{!*} fail, q_1 \xrightarrow{!*} fail$$

$$r_2 : q_1 \xrightarrow{?e(\alpha), l} q_2 \notin \{pass, fail\} \vdash q_1 \xrightarrow{?e(\alpha), l} q_2, q_1 \xrightarrow{!*} fail$$

$$r_3 : q_1 \xrightarrow{!e(\alpha), l} q_2 \vdash q_1 \xrightarrow{!e(\alpha), l} q_2, q_1 \xrightarrow{!*} inconclusive$$

$$r_4 : q_1 \xrightarrow{!e(\alpha), l} q_2, q_1 \xrightarrow{?e(\alpha), l} q_3 \not\in \rightarrow \vdash q_1 \xrightarrow{\theta} fail$$

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 !*) 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 $q_1 \xrightarrow{!*} inconclusive$ 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 $t = < e_1(\alpha_1), l_1 > \cdots < e_k(\alpha_k), l_k > \in ATraces$ do
2      Choose arbitrary $\sigma \in cl(t)$ ;
3      $cl(t) := cl(t) \setminus \{\sigma\}$;
4      $tc := lts(t, \sigma, v(t))$;
5      foreach $t_2 \in ATraces, \sigma_2 \in cl(t_2)$ such that
         $\exists \sigma_p \in \bigcup_{\sigma \in traces(tc)} prefix(\sigma) \cap prefix(\sigma_2) : out(\sigma, \sigma_p) \neq$
         $out(\sigma_2, \sigma_p)$ do
6          $tc_2 := lts(\sigma_2, v(t_2))$;
7          $cl(t_2) := cl(t_2) \setminus \{\sigma_2\}$;
8          $tc := tc \| tc_2$;
9          $ATraces := ATraces \setminus \{t_2\}$;
10     $tc := compl(tc)$;
11     complete the input events of $tc$;
12     $TC := TC \cup \{tc\}$;
13     $ATraces := ATraces \setminus \{t\}$;
```

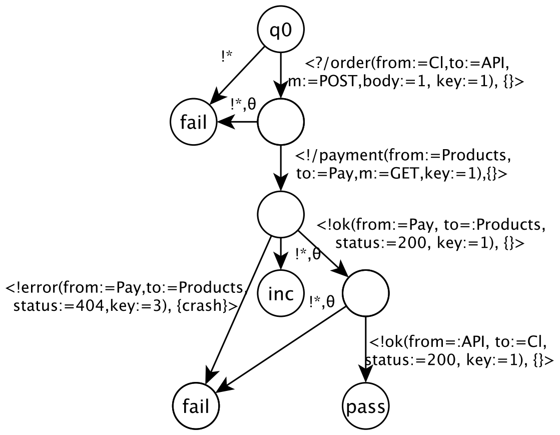

q0
!*
<?/order(from:=Cl,to:=API,
m:=POST,body:=1, key:=1), {}>
!*,θ
fail
<!/payment(from:=Products,
to:=Pay,m:=GET,key:=1),{}>
<!ok(from:=Pay, to=:Products,
status:=200, key:=1), {}>
!*,θ
<!error(from:=Pay,to:=Products,
status:=404,key:=3), {crash}>
inc
!*,θ
<!ok(from=:API, to:=Cl,
status:=200, key:=1), {}>
fail
pass
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" ($PO($"$Products, e()\alpha) : false$). The events /supply()

and the response ok() were hence removed from the test case to comply with the new test architecture.

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 $q_1 \xrightarrow{?req(\alpha),l} q_2$ and its response $q_1 \xrightarrow{!resp(\alpha'),l'} q_2$. These nested events have to be taken into account while the test case mutation. For instance, for Event Removal, if the operator deletes $?req(\alpha)$ and its response into a test case, it also needs to delete the potential nested requests and responses $!e_1(\alpha_1),\dots,!e_k(\alpha_k)$ such that $to(?req(\alpha)) = from(!e_1(\alpha_1)), to(!e_i(\alpha_i)) = from(!e_{i+1}(\alpha_{i+1}))(1 \leq i < k)$ and $to(!e_k(\alpha_k)) = from(!resp(\alpha'))$.

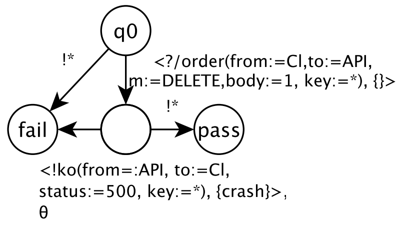

q0
!*
<?/order(from:=Cl,to:=API,
m:=DELETE,body:=1, key:=*), {}>
!*
fail
pass
<!ko(from=:API, to:=Cl,
status:=500, key:=*), {crash}>,
θ

Figure 7: Test case example obtained with the mutation operator "HTTP Verb Change"

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 $p := *$. 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 produces test cases written with the Citrus framework[2]. A complete evaluation will be presented in a future work.

REFERENCES

Aouadi, M. H. E., Toumi, K., and Cavalli, A. R. (2015). An active testing tool for security testing of distributed systems. In *10th International Conference on Availability, Reliability and Security, ARES 2015, Toulouse, France, August 24-27, 2015*, pages 735–740. IEEE Computer Society.

Beschastnikh, I., Brun, Y., Ernst, M. D., and Krishnamurthy, A. (2014). Inferring models of concurrent systems from logs of their behavior with csight. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 468–479, New York, NY, USA. ACM.

Cao, D., Felix, P., Castanet, R., and Berrada, I. (2009). Testing Service Composition Using TGSE tool. In Press, I. C. S., editor, *IEEE 3rd International Workshop on Web Services Testing*

[2]https://citrusframework.org

Name	Def.	Desc.: all the test cases that can be derived from tc by	Expected
Event Duplication	$\forall(?e(\alpha),L) \in \Sigma_I$, $add(tc,(?e(\alpha),L),(?e(\alpha),L))$	adding just before $?e(\alpha)$ a copy of the event followed by its related response	at least the response, no error, no crash
Event Swapping	$\forall(?e_1(\alpha_1),L_1),(?e_2(\alpha_2),L_2) \in \Sigma_I$, $swap(tc,(?e_1(\alpha_1),L_1),(?e_2(\alpha_2),L_2))$	swapping two different input events and their related responses along with all their nested output events	no crash
Event Removal	$\forall(?e(\alpha),L) \in \Sigma_I$, $del(tc,(?e(\alpha),L))$	deleting an input event and its related response along with all the nested output events. tc must still hold at least one input event	no crash
HTTP Verb Change	$\forall(?e(\alpha),L) \in \Sigma_I$, $Verb(tc,(?e(\alpha),L))$	changing the HTTP verb of the input event if available	no crash
Data Alteration	$\forall(?e(\alpha),L) \in \Sigma_I$, $dataAlt(\alpha \setminus \{from := c_1, to := c_2, session\_id := id1, token := id2\})$	randomly changing the data	no crash
Delay Addition	$\forall(?e(\alpha),L) \in \Sigma_I$, $add(tc,(?e(\alpha),L),\theta)$	adding a delay during which no reaction should be observed before every input event	no error, no crash
Token Removal	$\forall(?e(\alpha),L) \in \Sigma_I, "token" \in L :$ $delToken(tc,\alpha)$	delete the token in the input event	error, no crash
Token Alteration	$\forall(?e(\alpha),L) \in \Sigma_I, "token" \in L :$ $tokenAlt(tc,\alpha)$	replacing a token by another one	error, no crash
Session id Alteration	$\forall(?e(\alpha),L) \in \Sigma_I :$ $sessionidAlt(tc,\alpha)$	replacing a session id by another one	error, no crash
Session Closure	$closingSession(tc)$	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	error, no crash
Stress Testing	$\forall(?e(\alpha),L) \in \Sigma_I$, $add_{100}(tc,(?e(\alpha),L))$	adding just before $?e(\alpha)$ 100 copies of the event with overwhelming data	no error, no crash

Table 1: Mutation operators and short description

(WS-Testing 2009), Los Angeles, United States. IEEE Computer Society Press.

Hierons, R. (2001). Testing a distributed system: generating minimal synchronised test sequences that detect output-shifting faults. *Information and Software Technology*, 43(9):551–560.

Kanso, B., Aiguier, M., Boulanger, F., and Touil, A. (2010). Testing of Abstract Components. In *ICTAC 2010 - International Conference on Theoretical Aspect of Computing.*, pages 184–198, Brazil.

Köroglu, Y. and 0001, A. S. (2018). TCM: Test Case Mutation to Improve Crash Detection in Android. In *Proceedings of the 21st International Conference on Fundamental Approaches to Software Engineering*, pages 264–280. Springer.

Mariani, L. and Pastore, F. (2008). Automated identification of failure causes in system logs. In *Software Reliability Engineering, 2008. ISSRE 2008. 19th International Symposium on*, pages 117–126.

Paiva, A., Restivo, A., and Almeida, S. (2020). Test case generation based on mutations over user execution traces. *Software Quality Journal*, 28.

Petrenko, A. and Avellaneda, F. (2019). Learning communicating state machines. In *Tests and Proofs*, page 112–128, Berlin, Heidelberg. Springer-Verlag.

Phillips, I. C. C. (1987). Refusal testing. *Theor. Comput. Sci.*, 50:241–284.

Salva, S. and Blot, E. (2020). Cktail: Model learning of communicating systems. In Ali, R., Kaindl, H., and Maciaszek, L. A., editors, *Proceedings of the 15th International Conference on Evaluation of Novel Approaches to Software Engineering, ENASE 2020, Prague, Czech Republic, May 5-6, 2020*, pages 27–38. SCITEPRESS.

Salva, S., Provot, L., and Sue, J. (2021). Conversation extraction from event logs. In Cucchiara, R., Fred, A. L. N., and Filipe, J., editors, *Proceedings of the 13th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management, IC3K 2021, Volume 1: KDIR, Online Streaming, October 25-27, 2021*, pages 155–163. SCITEPRESS.

Torens, C. and Ebrecht, L. (2010). Remotetest: A framework for testing distributed systems. In *2010 Fifth International Conference on Software Engineering Advances*, pages 441–446.

Ulrich, A. and König, H. (1999). *Architectures for Testing Distributed Systems*, pages 93–108. Springer US, Boston, MA.

van der Bijl, M., Rensink, A., and Tretmans, J. (2004). Compositional testing with ioco. In Petrenko, A. and Ulrich, A., editors, *Formal Approaches to Software Testing*, pages 86–100, Berlin, Heidelberg. Springer Berlin Heidelberg.