# Automated Test Case Generation for Service Composition from Event Logs

Sébastien Salva and Jarod Sue
LIMOS - UMR CNRS 6158
University Clermont Auvergne, France
email: sebastien.salva@uca.fr, jarod.sue@uca.fr

***Abstract*—Service compositions, e.g., Internet of Things (IoT) compositions or RESTful service compositions are widely used in the industry to enhance the interoperability and integration of their systems and applications. Testing service compositions is long and difficult activity as each service may be deployed on different servers and often requires specialised testing tools. In order to help developers in this activity, this paper proposes an automated approach to generate test cases for experimenting every service in isolation. These test cases can be later adapted or used for regression testing. This approach is based upon 4 steps that aim to: 1. extract traces from event logs, 2. gather similar behaviours to reduce the final test case number and to extract knowledge that will be used during the test case generation, 3. produce generic test cases given under the form of IOTS (Input Output Transition Systems) that encode the use of mock components and provide test verdicts, 4. generate test scripts and mock components for every testable service. We report experimental results obtained from 4 case studies and show that our algorithms build effective test cases and scale well with the event log size.**

***Index Terms*—Service Composition; Test Case Generation; Mock Generation; Event Log***

## I. Introduction

Events that occur in service compositions are now commonly recorded in log files. These files are more and more analysed with tools allowing to continuously extract knowledge helping IT personnel understand system behaviours or performance. In this paper, we propose to use event logs to automatically generate tests for service compositions. In this scope, it is well admitted that the design and use of automatic test generation approaches is quite interesting in the industry since the testing of service compositions is usually performed manually by writing test scripts. The design of automatic approaches is also quite challenging as testing such systems is known to be a hard and long process due to the problems inherent in controlling or monitoring many concurrent components interacting with one another simultaneously.

The literature offers several approaches that might be considered to generate test cases for a service composition under test, which we denote SUT. Firstly, several Model based Testing (MbT) approaches e.g., [1, 2, 3, 4, 5, 6], have been proposed. Prior to test generation, a formal specification must be written and verified. The main problem inherent in these approaches is that such models are often not available or not up-to-date. A combination of model learning followed by test case generation might also be considered. Model learning is a research field gathering algorithms specialised to the construction of models by inference [7]. Once models are retrieved, a classical MbT approach must be applied to produce test cases. Test data along with concrete execution paths are re-generated from models. Basically, this combination starts from concrete events, such as event logs or observations obtained by experimentations, generalises them with formal models and re-generates concrete test cases. It results in a time consuming process, which usually does not scale well. Finally, other approaches, e.g., [8], propose a record and replay technique of event sequences extracted from event logs. Unfortunately, this technique does not work well for service compositions, as SUT may include various possibly non deterministic services, or not testable ones. Test cases must be adapted w.r.t. these properties.

We present in this paper another test case generation approach for service compositions from event logs, which is devised with the previous attention points in mind. The resulting test cases aim at experimenting every testable service of SUT in isolation. Unlike the previous approaches, this implies that our approach generates test scripts with verdicts and *mock components* from event logs. Mocking objects or components is a technique applied to improve the ability of interaction with the component under test, which finally aims to increase test coverage or to speed up performance. With service compositions, our generated mock components aim at simulating real services, and will be used to simplify the dependencies that make testing difficult. Furthermore, mock components should increase test efficiency by replacing slow-to-access components.

Given an event log collected from SUT, our approach performs four main steps. The first one converts an event log into formatted event sequences called traces. The latter are then gathered into clusters of similar traces, from which some knowledge is extracted by means of an expert system. This knowledge will be used to build test cases along with test verdicts. Then, test cases modelled with IOTS (Input Output Transition systems) are generated for every service. An IOTS test case expresses both the behaviour of a tester and the behaviours of mock components. Concrete test scripts and mock components are eventually derived from IOTS test cases.

This paper also provides an empirical evaluation, which investigates the effectiveness of our algorithms by assessing the quality of the generated test cases, and the performance in

terms of execution times. The implementation of the approach specialised for web service compositions is publicly available in [9].

**Paper organisation:** Section II discusses related work. Section III introduces our approach with an example used throughout the paper, along with the related work. Section IV provides some definitions, used by our algorithms, which are detailed in Section V. The next section exposes our evaluation performed on 4 service compositions. Section VII summarises our contributions and draws some perspectives for future work.

## II. Related work

A plethora of works have been proposed to generate test cases without specification, by using random testing or model learning. A few of them are specialised to communicating systems e.g., [10, 11, 12, 13, 14]. For instance, Arcuri proposed algorithms to create test suites for Web service compositions by considering the test case generation as a multi-objective problem, whose objectives are related to metrics over source code properties. EVOMaster [11] implements this algorithm to generate robustness tests still for Web service compositions. Unfortunately, this kind of white-box approach requires the source code of the system to be accessible. We do not consider this requirement in our approach. Instead of apply random testing, the approach presented in [13] recovers, through active model learning, a model $M_R$ of a reference implementation $R$, which serves as input for a model based testing tool. The obtained test cases are used for regression testing or to check whether another implementation $I$ conforms to $M_R$. Petrenko et al. improved this technique by relaxing some requirements, e.g., the components may be unknown. [12]. The tests, produced while the recovery of the component behaviours could be adapted to answer our problem. But this approach is founded upon some assumptions that strongly limit its adoption (the system should produce only a single message at a time, it cannot be composed of concurrent components, all of the components have to be testable). Zhang et al. proposed a test case generation approach for Web applications from event logs in [14]. The logs are initially covered to infer a Markov chain whose states are labelled by URLs. A MbT technique is then applied on this model. Specifically, test cases are built by covering the branches of the Markov chain. Generally speaking, we observed that model learning associated to MbT tends to be time consuming, as logs or test results are lifted to the level of models, and are then analysed to build again concrete test cases. None of these approaches allows the generation of mock components specialised for services.

## III. Overview

We introduce, in this section, a motivating example along with the presentation our algorithms at a high level. The details of the test case generation are given in Section IV. Our example, illustrated in Figure 1, is made up of 4 web services providing approvals or rejects of loan requests. A first service "LoanApp" receives a loan request linked to a given account. According to the amount requested, it calls a second service "AccMan" to get access to the bank account. This service calls itself "CheckRisk" to obtain the risk level related to this account. If the amount is upper than 10000 euros or if the risk is high, a third service "AppMan" is requested to let a human agent study the request and return a response. Otherwise "LoanApp" returns a positive response.

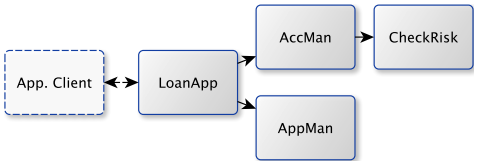


Fig. 1. Example of composition made up of 4 web services

### A. Assumptions

The design of our algorithms is guided by the following practical assumptions:

- **A1 Event log:** event logs are collected in a synchronous environment made up of synchronous communications. They include timestamps showing when the events occurred. For simplicity, we consider having one event log;
- **A2 Event content:** services produce communication events or non-communication events. The former include parameter assignments allowing to identify the source and the destination of each event. Besides, a communication event can be identified either as a request or a response;
- **A3 Service collaboration:** work-flows of events are correlated by means of parameter assignments. The parameter set is called correlation set;
- **A4 Testable Service:** we assume knowing the set of services that can be experimented and monitored, which is denoted *PCO*.

### B. Approach overview

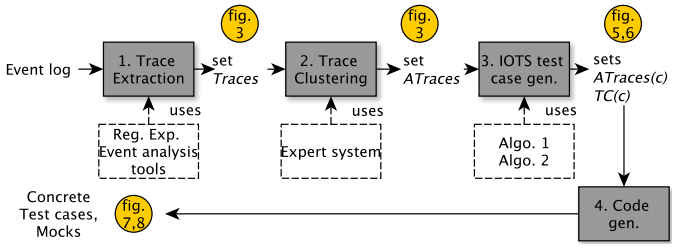


Fig. 2. Approach outline

Our approach is organised into four-steps, illustrated in Figure 2. The examples used in the following description are also referred in the figure. The user initially gives as input an event log collected from SUT, and finally gets executable test cases along with *mock components* to experiment every testable service of SUT in isolation.

The first step converts an event log into a set *Traces* of formatted event sequences, which intuitively correspond to sequences of correlated events interchanged among different services. Several tools or algorithms are available in the literature to perform this task. We studied them and presented a tool set in [15]. Consider the two traces of Figure 3 extracted

/askLoan(from:=Client , to :=LoanApp,method:=POST,body:=1000,acc:=99,id:=∗)
/checkAccountRisk(from:=LoanApp,to:=AccMan,method:=GET,acc:=99,id:=∗)
/ evaluateRisk (from:=AccMan,to:=CheckRisk,method:=GET,acc:=99,id:=∗)
/ok(from:=CheckRisk,to:=AccMan,method:=GET,body:=HIGH,status:=200,id:=∗)
/ok(from:=AccMan,to:=LoanApp,method:=GET,status:=200;id:=∗)
/checkApp(from:=LoanApp,to:=AppMan,method:=GET,acc:=99,body:=1000,id:=∗)
/ok(from:=AppMan,to:=LoanApp,method:=GET,status:=200,body:=ko,id:=∗)
/ rejectLoan (from:=LoanApp,to:=AccMan,method:=GET,acc:=99,id:=∗)
/ok(from:=AccMan,to:=LoanApp,method:=GET,status:=200,body:=Rejected,id:=∗)
/ok(from:=LoanApp,to:=Client,method:=GET,status:=200,body:=Rejected, id :=∗)

/askLoan(from:=Client , to :=LoanApp,method:=POST,body:=1000,acc:=99,id:=∗)
/checkAccountRisk(from:=LoanApp, to:=AccMan,method:=GET,acc:=99,id:=∗)
/ evaluateRisk (from:=AccMan,to:=CheckRisk,method:=GET,acc:=99,id:=∗)
/ok(from:=CheckRisk,to:=AccMan,method:=GET,body:=LOWRISK,status:=200,id:=∗)
/ok(from:=AccMan,to:=LoanApp,method:=GET,status:=200,id:=∗)
/acceptLoan(from:=LoanApp,to:=AccMan,method:=GET,body:=1000,acc:=99,id:=∗)
/ko(from:=AccMan,to:=LoanApp,method:=GET,status:=500,body:=ServerError,id:=∗)
/ok(from:=LoanApp,to:=Client,method:=GET,status:=200,body:=ServerError , id :=∗)

Fig. 3. Example of two traces collected from the composition of Figure 1

from the web service composition of Figure 1, made up exclusively of testable services. The first trace $t_1$ results from the request of a small amount loan. We can observe that AccMan is called and returns a high risk. AppMan is then called to get a response from a bank agent. Here, the loan is rejected. The parameter *id* used to correlate events is hidden (assigned to "*") in order to start generalise the behaviours found in traces. The second trace $t_2$ follows the same scenario, but a low risk is returned. While a request /acceptLoan is performed to update the account, a server error occurs. An error is returned to the client.

</askLoan(from:=Client,to:=LoanApp,method:=∗,body:=∗,acc:=∗,id:=∗){}>
</checkAccountRisk(from:=LoanApp,to:=AccMan,method:=∗,acc:=∗,id:=∗),{}>
</evaluateRisk(from:=AccMan,to:=CheckRisk,method:=∗,acc:=∗,id:=∗),{}>
</ok(from:=CheckRisk,to:=AccMan,method:=∗,body:=∗,status:=∗,id:=),{}>
</ok(from:=AccMan, to:=LoanApp,method:=∗,status:=∗,id:=∗),{}>
</acceptLoan(from:=LoanApp,to:=AccMan,method:=∗,body:=∗,acc:=∗,id:=∗),{}>
</ko(from:=AccMan,to:=LoanApp,method:=∗,status:=∗,body:=∗,id:=∗),{error}>
</ok(from:=LoanApp,to:=Client,method:=∗,status :=∗,body:=∗,id :=∗),{}>

Fig. 4. Example of abstract trace

Step 2 of our approach gathers the similar traces into clusters. Two similar traces share the same sequence of events and are performed by the same services. This step is performed to avoid the generation of large test case sets, as many similar traces, composed of the same event sequence accompanied by different parameters, may be found in event logs. With our previous example of two traces, we get two clusters $cl(t_1)$ and $cl(t_2)$. The clusters are then analysed with an expert system to extract some knowledge. At the moment, we try to extract the fact that an event represents an error or a failure, the fact that an event sequence represents a login process or a token generation. This knowledge, shortened under the form of labels, will be used for the generation of tests and of test verdicts. These clusters and labels are assembled to form *abstract traces*, which correspond to sequences of elements $< e(\alpha), l >$, with $e(\alpha)$ an event whose parameter values are hidden and $l$ a list of labels expressing some knowledge. With our example, 2 abstract traces are built. Figure 4 gives the abstract trace of the second trace of Figure 3. The seventh event is recognised as an error.

<?/checkAccountRisk(from:=LoanApp,to:=AccMan,method:=∗,acc:=∗,id:=∗),{}>
<!/evaluateRisk(from:=AccMan,to:=CheckRisk,method:=∗,acc:=∗,id:=∗),{mock}>
<!/ok(from:=CheckRisk,to:=AccMan,method:=∗,body:=∗,status:=∗,id:=),{mock}>
<!/ok(from:=AccMan,to:=LoanApp,method:=∗,status:=∗,id:=∗),{}>
<?/acceptLoan(from:=LoanApp,to:=AccMan,method:=∗,body:=∗,acc:=∗,id:=∗),{}>
<!/ko(from:=AccMan,to:=LoanApp,method:=∗,status:=∗,body:=∗,id:=∗),{error}>

Fig. 5. Example of abstract trace for the service AccMan

From the set of abstract traces denoted *ATraces*, Step 3 builds test cases. A test case encodes interactions between a tester and one service under test, along with the interactions of this service with other dependee services. In our context of isolation testing, the dependee services have to be replaced with mock components. To generate test cases and mock components, Step 3 starts by building the abstract trace set $ATraces(c)$ of every testable service $c$. In the meantime, the algorithm decorates the events with the symbols ? and ! to express the notion of input and output. It also adds the label "mock" to the events produced by the dependee services. These specific events will be used to generate mock components. To avoid ambiguity, it is worth noting that an input (resp. output) refers to the inputs (resp. outputs) of a service under test, that is what it expects (returns). Figure 5 shows an abstract trace example for the service AccMan.

Step 3 then generate test cases, given under the form of IOTS trees, for every testable component. The use of the IOTS formalism allows to synthesize generic test cases from which can be derived concrete test scripts. The resulting test cases encode the interactions (inputs) that can be performed, all the different behaviours that can be observed and the respective test verdicts. These test cases are built w.r.t. the following restrictions to ensure that they can be executed: at most one input is doable at every state of a test case and any output may be observed. Additionally, the next restriction aims at limiting the number of output events that may performed by mock components: at every state, one output event labelled by "mock" is allowed only. The test verdicts are given by means of the labels found within the abstract traces. Intuitively, the test verdict is fail if the label "error" is found with the last output event. Otherwise, the verdict is pass. Figure 6 illustrates an IOTS test case for the service AccMan, obtained from Figure 5. A fail verdict is given as the event !ko corresponds to an error. Besides, if no reaction is observed whereas an output is expected, it returns fail. For readability, this corresponds to the dotted transitions labelled with the symbol θ. If unexpected outputs are received (dashed transitions labelled by "!*"), it returns the verdict inconclusive, meaning that we cannot conclude. We cannot definitely conclude because the event log may not include all the possible behaviours that can be performed by a component.

Finally, Step 4 converts every IOTS test case into test scripts. All the transitions labelled by "mock" are put aside. The remaining tree is converted into a test script. We exemplify this step with the frameworks TESTNG [1] and Citrus [2], both

[1]https://testng.org
[2]https://citrusframework.org/

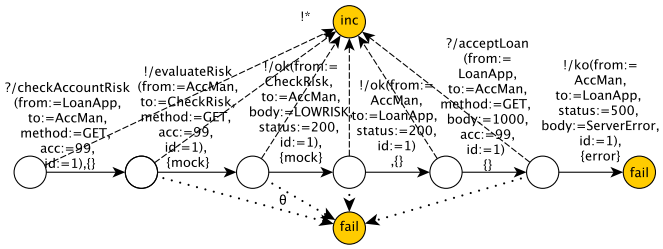


Fig. 6. IOTS Test Case for AccMan

based upon JUNIT. The later helps testers write test cases for services upon varied message protocols, e.g., HTTP or TCP/IP. The beginning of the test case related to the IOTS of Figure 6 is given in Figure 7. The service AccMan is called with the request "checkAccountRisk". The test case then asserts that a HTTP response is received with the status code 200. A valid response includes only the session identifier. The test ends by checking with verificationMock() whether the mock components have been called the expected number of times.

On the other hand, the IOTS transitions labelled by "mock" are used to generate mock components, which are composed of rules of the form *"request() ... response()"*. For every request labelled by mock from a service $c_1$ to $c_2$, we search for the next response from $c_2$ to $c_1$ and we build one rule. Figure 8 lists one rule of the mock "CheckRisk", which is written with the framework Mockserver [3]. These steps and algorithms are now detailed in the following.

```
@Test @CitrusTest
public void testAccMan() throws FileNotFoundException{
HttpClient toClient = CitrusEndpoints
. http () . client () . requestUrl (" http :// AccMan/").build() ;
$( http ()
. client ( toClient ) . send() . get ("checkAccountRisk").message()
.header(" id ",1) . body("\"acc\"=99").accept(MediaType.ALL_VALUE));
$( receive ( toClient )
.message() . type(MessageType.PLAINTEXT).name("Response").extract(fromHeaders()
.header(HttpMessageHeaders.HTTP_STATUS_CODE, "statusCode"))
.header(" id "," id"))) ;
variable ("body"," citrus :message(Response.body())") ;
variable (" status ", "${statusCode}");
String body = context . getVariable ("body");
String status = context . getVariable (" status ") ;
String id = context . getVariable (" id") ;
If (body.equals ("") && id.equals("1") && status.equals ("200")) assertTrue ( true ) ;
else Assumptions.assumeTrue(false ," Inconclusive ") ;
...
verificationMock () ; }
```

Fig. 7. Example of test script for the service AccMan

```
mockServer.when(
request () .withMethod("GET").withPath("/EvaluateRisk")
.withHeaders( new Header("acc", "99"),new Header("id", "1"))
,Times.exactly (1) )
.respond(
response ()
.withHeaders(new Header("id", "1")) .withStatusCode(200)
.withBody("LOWRISK"));
```

Fig. 8. Mock component piece of code, which implements the events !/EvaluateRisk and !ok of the test case of Figure 6

[3]https://www.mock-server.com

## IV. Test Case and Mock Component Generation

In our context of service composition, we consider that events have the form $e(\alpha)$ with $e$ some label and $\alpha$ an assignment of parameters in $P$ to a value in the set of values $V$. These parameters allow the encoding of some specific features for service compositions e.g., if an event is a request, the receiver and sender of this request, etc. We write $x := *$ the assignment of the parameter $x$ with an arbitrary element of $V$, which is not of interest. $\mathcal{E}$ denotes the event set. The concatenation of two event sequences $\sigma_1$, $\sigma_2$ is denoted $\sigma_1.\sigma_2$. $\varepsilon$ stands for the empty sequence. For sake of readability, $prefix(\sigma)$ denotes the set of initial segments of $\sigma$ and we write $\sigma_1 \in \sigma_2$ iff $\sigma_1 \in prefix(\sigma_2)$. We also use additional notations on events to make our algorithms more readable. In particular, the notation $deps(e(\alpha))$ returns the dependent service involved in the exchange of the event $e(\alpha)$ with some dependee service:

**Definition 1** *Let $e(\alpha)$ be an event of $\mathcal{E}$.*

- *$from(e(\alpha)) = c$ denotes the source component performing $e(\alpha)$;*
- *$to(e(\alpha)) = c$ denotes the destination;*
- *$isreq(e(\alpha))$, $isresp(e(\alpha))$ are boolean expressions expressing the nature of the event;*
- $deps(e(\alpha)) = \begin{cases} from(e(\alpha)) & \text{iff } isreq(e(\alpha)) \\ to(e(\alpha)) & \text{iff } isresp(e(\alpha)) \\ * & \text{otherwise} \end{cases}$

A test case is a deterministic IOTS having a tree form and whose sink states are either pass, fail or inconclusive. IOTS transitions are given under the form $q \xrightarrow{e(\alpha),l} q'$ with $e(\alpha)$ some event and $l$ a label set, which may be empty. Furthermore, we use the notation $\theta$ labelled on transitions to represent the absence of reaction from a service under test.

IOTS test cases should be constructed with a few restrictions to avoid indeterministic behaviours while testing. To this end, a test case should be deterministic and should allow at most one input event at any state. In reference to [16], we formulate this last restriction by saying that a test case is *input restricted*. Additionally, still in the context of isolation testing and to keep control of the testing process, a mock component should return at most one response after being invoked with the same event. As a consequence, test cases should also have states that offer at most one output expressing a response. We say that a test case is mock response restricted. This is formulated with:

**Definition 2** *A test case $tc$ is a deterministic IOTS $\langle Q, q0, \Sigma \cup \{\theta\}, L, \rightarrow \rangle$ where:*

- *$Q$ is a finite set of states; $q0$ is the initial state; $Q$ contains three special states: pass, fail and inconclusive*
- *$\Sigma$ 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 $\Sigma_O \cap \Sigma_I = \emptyset$*
- *$L$ is a set of labels*
- *$\rightarrow \subseteq Q \times \Sigma \cup \{\theta\} \times L^* \times Q$ is a finite set of transitions. A transition $(q, e(\alpha), l, q')$ is also denoted $q \xrightarrow{e(\alpha),l} q'$*

- *tc has no cycles*
- *tc is input restricted i.e.* $\forall q \in Q : event(q) = \Sigma_O \cup \{e(\alpha)\}$ *for some* $e(\alpha) \in \Sigma_I$ *or* $event(q) = \Sigma_O \cup \{\theta\}$ *with* $event(q) = \{e(\alpha) \mid \exists q' \in Q : q \xrightarrow{e(\alpha),l} q'\}$
- *tc is mock response restricted i.e.* $\forall q \in Q : |\{q \xrightarrow{e(\alpha),l} q' \mid isResp(e(\alpha)) \wedge mock \in l\}| \leq 1$.

## V. Test Case Generation

### A. Step 1: Trace Extraction

We do not focus on this step in this paper and refer to [15] instead. In short, the tool proposed in the paper returns the set *Traces* along with a correlation set $Corr(\sigma)$ for every trace $\sigma \in Traces$. All the assignments $p := v$ found in $\sigma$ whose parameters are also used in $Corr(\sigma)$ are replaced by $p := *$.

### B. Step 2: Trace Clustering

This step takes as input the set *Traces* 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 sets expressing some business knowledge about the events.

**Definition 3 (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* $l_i \subseteq L (1 \leq i \leq k)$.

Abstract traces ar extracted by partitioning the set *Traces* into equivalent classes. Two traces are said equivalent when 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 traces is defined by means of a projection, which performs this event abstraction:

**Definition 4** *Two event sequences* $\sigma_1$, $\sigma_2 \in \mathcal{E}^*$, *are equivalent, denoted* $\sigma_1 \sim_b \sigma_2$ *iff* $proj_{\{from,to\}}\sigma_1 = proj_{\{from,to\}}\sigma_2$ *with:* $proj_Q : \mathcal{E}^* \to \mathcal{E}^*$ *is the projection* $e_1(\alpha'_1) \ldots e_k(\alpha'_k) = proj_Q(e_1(\alpha_1) \ldots e_k(\alpha_k))$ *and* $\alpha'_i = \{x := * \mid x := v \in \alpha_i \wedge x \notin Q\} \cup \{x := v \mid x := v \in \alpha_i \wedge x \in Q\}$

The equivalent classes $\{cl_1, \ldots, cl_n\}$ are derived with $\sim_b$. Given a class $cl = \{\sigma_1, \ldots, \sigma_m\}$, our algorithm analyses the events and parameter values to extract knowledge by means of an expert system. Generally speaking, the latter is an inference engine that applies a set of rules to infer new facts. In our context, we devised rules to encode expert knowledge about service compositions and to build abstract traces. It is worth noting that an expert system offers the benefit to save time by allowing its reuse on several service compositions.

We represent inference rules with this format: *When conditions on facts Then actions on facts*. To ensure that this step is performed in a deterministic way, the inference rules have to be Modus Ponens (simple implications that lead to sound facts if the original facts are true).

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

Fig. 9. Inference rule example

We devised inference some rules that analyse event content or event sequences to recognise errors (bas status, crashes, etc.). Figure 9 exemplifies a rule for recognising a server crash by means of the HTTP status. It creates an abstract event decorated with a new label "error". We also observed in many component systems, that the proper functioning of a component may initially require a login process or the generation of Access tokens. These initial behaviours are recognised with further rules, which create abstract events composed of the labels "login" or "token". We denote $ID \subset L = \{"login","token"\}$.

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 $n$ equivalent classes of traces $cl_1, \ldots, cl_n$, we obtain a set of $n$ abstract traces, which is denoted *ATraces*. Figure 4 illustrates an example of abstract trace including a label "error" added by means of the previous inference rule.

### C. Step 3: IOTS Test Case Generation

**Algorithm 1:** Component Atrace set gen.

**input :** $ATraces$
**output:** $ATraces(c_1), \ldots, ATraces(c_n)$
1 **foreach** $t = < e_1(\alpha_1), l_1 > \cdots < e_k(\alpha_k), l_k > \in ATraces$ **do**
2  $tinit := < e'_1(\alpha'_1), l'_1 > \cdots < e'_m(\alpha'_m), l'_m > \in t$ such that $l_i \cap ID \neq \emptyset$ ;
3  Foreach $c \in PCO, t(c) := tinit$; $cl(t(c)) := cl(t)$;
4  **for** $1 \leq i \leq k$ **do**
5  **if** $isreq(e_i(\alpha_i)) \wedge (to(e_i(\alpha_i) \in PCO)$ **then**
6  $t(to(e_i(\alpha_i)) := t(to(e_i(\alpha_i)).< ?e_i(\alpha_i), l_i >$;
7  **if** $deps(e_i(\alpha_i) \in PCO$ **then**
8  $t(deps(e_i(\alpha_i)) := t(deps(e_i(\alpha_i)).< !e_i(\alpha_i), l_i \cup \{mock\} >$;
9  **if** $isresp(e_i(\alpha_i)) \wedge (from(e_i(\alpha_i) \in PCO)$ **then**
10  $t(from(e_i(\alpha_i)) := t(from(e_i(\alpha_i)).< !e_i(\alpha_i), l_i >$;
11  **foreach** $t(c) \neq tinit$ **do**
12  Update $cl(t(c))$ w.r.t. $t(c)$;
13  **if** $\exists t'(c) \in ATraces(c) : t'(c) = t(c)$ **then**
14  $cl(t'(c)) := cl(t'(c)) \cup cl(t(c))$;
15  **else**
16  $ATraces(c) := ATraces(c) \cup \{t(c)\}$;

The IOTS test case generation is implemented by Algorithms 1 and 2. Algorithm 1 takes as input a set of abstract traces *ATraces* and returns a set of $ATraces(c)$ for every service $c \in PCO$ found in the events. To build these new sets, Algorithm 1 covers every abstract trace $t \in ATraces$ (line 1). As stated previously, the proper functioning of a service may initially require a login process or a token generation. Our algorithm firstly covers the abstract trace $t$ to extract a subsequence *tinit* encoding this initial behaviour. The later is recognised with events associated with some labels in *ID* (line 2). The new abstract traces generated from $t$ will all begin with *tinit*, which may be empty. Then, Algorithm 1 builds a new abstract trace $t(c)$ for every service $c$ found in $t$. Besides,

it inserts the notion of input and output: if the event is a request to a testable service $c \in PCO$ it decorates the event with "?" (line 5); if the event is a response from a testable service (line 9 ), or an event for or from a dependee service, it decorates the event with "!". For this last case, the label "mock" is also added to events. This label will be used later to generate mock components. Indirectly, this algorithm filters out the other events, i.e.. the non communicating events or the events performed by non testable services.

Finally, the algorithm updates the traces of the cluster $cl(t(c))$ by deleting the events that belonged to the initial abstract trace $t$ but are no more available in $t(c)$. $t(c)$ is added to the set $ATraces(c)$. If $t(c)$ was already in $ATraces(c)$, only the cluster $cl(t(c))$ is updated.

**Algorithm 2:** IOTS Test Case Generation

```
   input  : ATraces(c)
   output: TC(c)
1  AT = ATraces(c);
2  while AT ≠ ∅ do
3      Take t =< e1(α1),l1 > ··· < ek(αk),lk >∈ AT;
4      Choose arbitrary σ ∈ cl(t) ;
5      tc := lts(t,σ,v(t));
6      Corr(tc) := Corr(σ)
7      foreach σ2 ∈ cl(t2) : t2 ∈ ATraces(c) ∧ prefix(σ) ∩ prefix(σ2) ≠ ∅ do
8          tc2 := iots(σ2,v(t2));
9          tc2 := tc ‖ tc2;
10         if tc2 is input and mock response restricted then
11             tc := tc2; AT := AT \ {t2};
12     tc := compl(tc); TC(c) := TC ∪ {tc}; AT := AT \ {t};
```

Algorithm 2 now takes as input a set $Atraces(c)$ and produces an IOTS test case set $TC(c)$. Given an abstract trace $t \in Atraces(c)$, the algorithm selects some trace $\sigma$ of the cluster $cl(t)$ and builds an initial test case $tc$ composed of parameter values (lines 2-6). The IOTS $tc$ is derived by means of the operator $iots : (\mathcal{E} \times L^*)^* \times \mathcal{E}^* \times \{fail, pass\} \rightarrow IOTS$, which returns an IOTS $\langle Q, q0, \Sigma, \rightarrow \rangle$ 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 last event $< e_k(\alpha'_k), l_k >$. This test verdict denoted $v(< e_1(\alpha_1), l_1 > \cdots < e_k(\alpha_k), l_k >$ is fail iff "*error*" $\in l_k$, otherwise the verdict is pass. Thereafter, Algorithm 2 covers each abstract traces $t_2 \in Atraces(c)$ and each trace $\sigma_2 \in cl(t_2)$ that shares some prefix with the initial trace $\sigma$ (lines 7-12). Intuitively, the trace $\sigma_2$ starts with a same event sequence than the test case $tc$ but may end with other events, which encode other behaviours. In this case, $tc$ must be completed to include those behaviours that may happen while testing. Algorithm 2 generates the IOTS $tc_2$ from $\sigma_2$ and performs a parallel synchronisation between $tc$ and $tc_2$. If the resulting test case is input restricted and mock response restricted, it meets the restrictions formulated in Definition 2. In this case, this new test case is assigned to $tc$. Additionally, as event logs do not necessarily encode all the behaviours of SUT, the test case $tc$ is completed (line 13) with the operator $compl : IOTS \rightarrow IOTS$ defined by these rules:

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

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

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

$r_4 : q_1 \xrightarrow{!e(\alpha),l} q_2, q_1 \xrightarrow{?e(\alpha),l} q_3 \notin \rightarrow \vdash q_1 \xrightarrow{\theta} fail$

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 $\theta$ is added to formulate that the absence of event is expected. Two transitions to inconclusive are added to express that we cannot conclude whether the behaviour is correct when we observe any other unexpected output event (label !*). $r_2$ targets the remaining transitions labelled by input events and similarly adds transitions to inconclusive. $r_3$ completes the test case with a new transition to express the fact that any unexpected output leads to the inconclusive verdict. $r_4$ 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 absence of reaction is faulty. These rules were applied to add the transitions to fail and inconclusive in the test case of Figure 6.

### D. Step 4: Generation of Concrete Test Cases

Finally, executable test scripts are generated from IOTS test cases. Different kinds of languages and frameworks may be chosen. With regard to our evaluation, we have chosen to generate test cases using the frameworks TestNG, Citrus and Mockserver. Given an IOTS test case $t \in TC(c)$, some parameters may still be assigned to "*". These ones refer to parameters used to identify sessions. We update these assignments with concrete values available in the correlation set $Corr(t)$. In case it still remains unassigned parameters, those are assigned with random values. In order to generate a test script from $t$, the transitions of $t$ labelled by "mock" are initially pruned. The resulting IOTS tree is converted into a TESTNG test case. In short, every input event is converted into code that calls the service under test $c$ and waits for a response. An example is given in Figure 7 (lines 1-20). The related transitions labelled by an output are used to build assertions. When there are several transitions expressing several correct responses, we use the word "AnyOf" to write an assertion that accepts several conditions. The test script ends with the call of the method "verificationMock", which aims to check whether mock components behave as expected while the test execution. At the moment, we check whether the number of calls to a mocked request matches with the number of time the request is found in $t$.

It remains to generate mock components. The previous IOTS transitions labelled by "mock" are used to derive rules of the form *request()...response()*. More precisely, for each request to a service $c2$ and its related response, a rule, which mimics the behaviour of $c2$, is constructed. Figure 8 shows an example of rule written with the language provided by the framework MockServer. Then, the method "verificationMock" is written according to these rules.

| Comp. | Event log size | # Test Cases | # Mutants | Mut. score | Mut. score 2 |
|---|---|---|---|---|---|
| C1 | 6440 | 61 | 146 | 0.96 | 0.96 |
| C2 | 1073 | 88 | 84 | 0.26 | 0.78 |
| C3 | 292 | 67 | 101 | 0.33 | 0.65 |
| C4 | 354 | 134 | 48 | 0.46 | 0.92 |

TABLE I
QUALITY EVALUATION OF THE TEST SUITES

## VI. PRELIMINARY EVALUATION

We implemented our approach for web service compositions and internet of things communicating over the HTTP protocol. With this implementation, we evaluated these questions:

- RQ1: what is the quality of the generated test suites ?
- RQ2: how long does our approach take to generate test suites? How our tool scales with the log size ?

The study was conducted on 4 web service compositions, denoted C1 to C4, made up of 4 to 6 components. We chose to consider different compositions in terms of code quality. We wrote C1 by refactoring and putting care into the code quality (no useless or duplicated code, strict parameter validation, use of design patterns). C2 to C4 were written by students and include useless getters, have improper error managements. From each composition, event logs were collected from scenarios performed by hands and completed by means of the penetration testing tool ZAP [4]. We obtained event logs composed of 292 to 6440 events to also consider the impact of the event log size. The source code in Java along with event logs are available in [9].

### A. RQ1: what is the quality of the generated test suites ?

To investigate RQ1, we firstly generated the test suites of every service C1 to C4. The test suite quality is evaluated with mutation testing. This software testing technique firstly performs small changes to the source code, which are called mutants. The later are then experimented with test cases. The mutants that are detected by test cases are said killed. The quality is measured by calculating the mutation score, obtained by dividing the number of killed mutants by the total number of mutants generated. A high mutation score indicates high test quality. We generated mutants from every web service with the tool PITest [5] completed by our own mutations specialised to Web services (Deletion of Authentication Token, Header removal, HTTP Verb change). Then, we experimented these mutants with the generated test cases to calculate mutation scores. The results are given in Table I, which provides the number of generated test cases, the number of mutants and the mutation score for C1 to C4. We obtain a high mutation score for C1 but passable results for C2 to C4. For these compositions, we observed that some mutants cannot be killed, i.e. they cannot be detected by our tests. The later are indeed built over communicating events found in event logs only. In other terms, every service is considered as a black box and test cases are not suited to detect local variable changes in the source code. But many useless local variables and prints in output console are used in C2 to C4. Besides, we observed that the removal of the verb "GET" produces non-killable mutants because when there is no verb in the service source code, then "GET" is used by default. This is why we chose to calculate a second score based upon the killable mutants only. These scores are now between 65 to 96%. We then analysed the killable mutants that are not detected by test cases. We observed that some mutations changed some parameter values. But these values are not used in test cases, hence the mutants were not killed. This problem comes from to the incompleteness of the event logs. Usually, event logs do not include all the possible behaviours (all the scenarios allowed in the real compositions). As a consequence, the generated test suite is not exhaustive and cannot detect all the possible mutants and faults. The more complete the event log is, the more exhaustive the test suite will be. This is especially the case for C3, whose event log includes only 292 events.This point tends to suggest that a supplementary criterion seems required to assess whether the event log is complete enough before starting test case generation.

### B. RQ2:how long does our approach take to generate test suites? How our tool scales with the log size ?

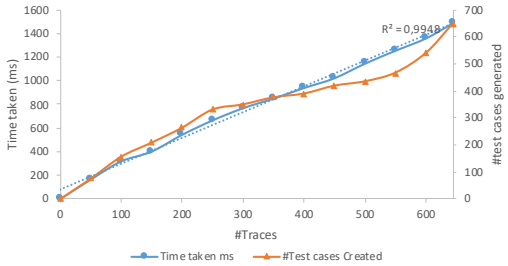


Fig. 10. Execution times vs. trace number

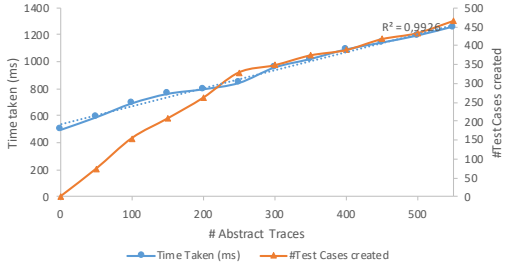


Fig. 11. Execution times vs. abstract trace number

The performance of our algorithms mainly depends on two factors, the size of the event logs (steps 1 and 2) and the number of abstract traces (steps 3 and 4). For the former factor, we took back the log of C1 composed of 6440 events, we extracted 650 traces and split them into sets of 50 to 650 traces. Then, we executed our tool to get execution times, which are given in Figure 10 in milliseconds. The test case generation took less than 2 seconds with the largest set. Figure 10 shows that the execution time curve increases linearly with respect to the trace set size. To avoid any bias, Figure 10 also illustrates the curve of the generated test cases (mocks included), which follows the same trend. The study of the

[4]https://www.zaproxy.org/
[5]https://pitest.org/

second factor was conducted by feeding our algorithms with sets of 50 to 550 abstract traces by 50 increments. These sets were constructed from 50 initial traces made up of ten events at most, whose parameter were modified. Figure 11 depicts execution times along with the number of generated test cases. Again, we observe that the time complexity is linear.

All these results tend to suggest that our tool can take large event logs and produce effective test cases in reasonable time.

### C. Threats to validity

We have identified 5 possible threats to the validity of this study. We identified 3 internal threats and 2 external threats. The first three ones are related to the event log sizes, the fault injection and the expert knowledge rules. The larger the event log, the more complete the test case set will be. For this study, we generated small to large event logs by manually executing the service compositions and by applying ZAP. To compute mutation scores, mutants are generated with PITest. This tool builds a lot of mutants, but unfortunately the tool is nor specialised for web services. To reduce this bias, we manually checked the mutants and applied some additional mutations. Test cases are built with labels inferred in Step 2 by means of inference rules. We assume that the rules are sufficient to infer all the correct labels for this evaluation. But more rules are required to generalise the approach. The external validity of our results is firstly threaten by the chosen case studies. Even though we considered very different web services and codes, it remains difficult to claim that our approach can be generalised. We need to apply our approach to further service compositions, made up of large service sets. One main issue that may happen here is the explosion of test cases and of mock components.The generalisation of our approach is also restricted by the requirements A1-A4. We need to investigate how these requirements could be relaxed in future work.

## VII. Conclusion

We proposed in this paper an automated test case generation for service compositions, from event logs. The originality of the approach resides in the fact that test cases along with mock components are generated for every testable service to test them in isolation. Besides, our approach extracts knowledge by means of an expert system to recognise some specific behaviours considered while the test case construction. We performed a preliminary evaluation and showed that the generated test cases are effective when event logs contain sufficient events, and that our algorithms scale well. As future work, we plan to extend these algorithms with specialised test case mutation operators to expand the initial test case set. In particular, we will propose specific operators for simulating some specialised security failures and for improving fault localisation.

## References


[1] A. Ulrich and H. König, *Architectures for Testing Distributed Systems*. Boston, MA: Springer US, 1999, pp. 93–108.

[2] D. Cao, P. Felix, R. Castanet, and I. Berrada, "Testing Service Composition Using TGSE tool," in *IEEE 3rd International Workshop on Web Services Testing (WS-Testing 2009)*, I. C. S. Press, Ed. Los Angeles, United States: IEEE Computer Society Press, Jul. 2009.

[3] C. Torens and L. Ebrecht, "Remotetest: A framework for testing distributed systems," in *5th International Conference on Software Engineering Advances*, 2010, pp. 441–446.

[4] B. Kanso, M. Aiguier, F. Boulanger, and A. Touil, "Testing of Abstract Components," in *ICTAC 2010 - International Conference on Theoretical Aspect of Computing.*, Brazil, Sep. 2010, pp. 184–198.

[5] M. H. E. Aouadi, K. Toumi, and A. R. Cavalli, "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*. IEEE Computer Society, 2015, pp. 735–740.

[6] R. Hierons, "Testing a distributed system: generating minimal synchronised test sequences that detect output-shifting faults," *Information and Software Technology*, vol. 43, no. 9, pp. 551–560, 2001.

[7] S. Ali, H. Sun, and Y. Zhao, "Model learning: A survey on foundation, tools and applications," 2018.

[8] A. Paiva, A. Restivo, and S. Almeida, "Test case generation based on mutations over user execution traces," *Software Quality Journal*, vol. 28, 09 2020.

[9] J. Sue and S. Salva, "Test case an mock generation tool," 2023. [Online]. Available: https://github.com/JarodSue/AutomatedTestGeneration

[10] B. K. Ozkan, R. Majumdar, and S. Oraee, "Trace aware random testing for distributed systems," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, oct 2019.

[11] A. Arcuri, "Restful api automated test case generation with evomaster," *ACM Trans. Softw. Eng. Methodol.*, vol. 28, no. 1, jan 2019.

[12] A. Petrenko and F. Avellaneda, "Learning communicating state machines," in *Tests and Proofs*. Berlin, Heidelberg: Springer-Verlag, 2019, p. 112–128.

[13] F. Aarts, H. Kuppens, J. Tretmans, F. W. Vaandrager, and S. Verwer, "Improving active mealy machine learning for protocol conformance testing," *Mach. Learn.*, vol. 96, no. 1-2, pp. 189–224, 2014.

[14] X. Tian, H. Li, and F. Liu, "Web service reliability test method based on log analysis," in *2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, 2017, pp. 195–199.

[15] S. Salva, L. Provot, and J. Sue, "Conversation extraction from event logs," in *13th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management, IC3K 2021*. SCITEPRESS, 2021, pp. 155–163.

[16] J. Tretmans, *Model Based Testing with Labelled Transition Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 1–38.