Modelling and testing of service compositions in partially open environments

Sébastien Salva

LIMOS - CNRS UMR 6158
Université d'Auvergne, Campus des Cézeaux,
Aubière, France
sebastien.salva@u-clermont1.fr

RÉSUMÉ. *Habituellement, les compositions de services Web sont testées en supposant que celles-ci sont exécutées dans un environnement ouvert dans lequel tous les messages échangés entre les services participant à la composition sont observable. Néanmoins, lorsque des services sont déployés dans des environnements partiellement ouverts comme les Clouds, cette hypothèse ne peut plus être maintenue. Cet article propose une méthode afin de tester si une composition de services est conforme à sa spécification en référence à la relation de test ioco et en considérant que les messages internes échangés entre les services sont masqués mais que ces mêmes services (ou des copies) peuvent être appelés directement. Les spécifications sont modélisées par des systèmes de transitions symboliques (STS) qui nous spécialisons vis-à-vis des services Web avec quelques notations et fonctions. Notre approche consiste à décomposer un ensemble de cas de test existants selon l'imbrication des opérations que nous formalisons par un facteur appelé le degré de dépendance. Puis, grâce à l'exécution du nouvel ensemble de cas de test, des fragments de trace (réactions observables) sont collectés puis réassemblés. Grâce à ces traces, il est ainsi possible de vérifier si l'implantation est ioco-conforme à la spécification. Notre approche est illustrée grâce à un exemple dérivé de l'application d'externalisation de carnets de santé pour les patients et médecins, actuellement en cours de développement par la société Orange Business Service.*

ABSTRACT. *Usually, Web service compositions are tested by assuming that these ones are executed inside an open environment where all the messages exchanged between the services participating to the composition are observable. Nevertheless, when services are deployed in partially open environments e.g., Clouds, this assumption cannot be sustained. This paper proposes a method to check whether a service composition is conforming to its specification*

Studia Informatica Universalis., pages 155 à 185

according to the ioco test relation, by considering that the internal messages exchanged between the services are hidden but that we can invoke each service directly (or an exact copy). Specifications are modelled with Symbolic Transition Systems (STS) that we specialize in Web services with some annotations and functions. Our approach consists in decomposing an existing test case set according to the operation interleaving that we formalize with a factor denoted the dependency degree. Then, while executing the new test case set, we recover fragments of traces (observable reactions) that are reassembled. With the final traces, we are able to check whether the implemented composition is ioco-conforming to its specification. We illustrate our approach with an example derived from the application of Electronic Health Record externalization for both patients and practitioners, currently in development by the Orange Business Service company.

MOTS-CLÉS : *composition; environnement partiellement ouvert; relation de test ioco; test de conformité; recomposition de traces*

KEYWORDS: *composition; partially open environment; ioco test relation; conformance testing; trace reconstruction*

---

1. Introduction

Component based software programming is a famous paradigm which is now pervasive and ubiquitous in most of software development companies. This one offers many advantages such as the building of software upon existing and heterogeneous components, the reuse of software accompanied by cost reduction, or the design of clear specifications of the inputs needed from other components.

This paradigm has also given rise to several recent trends and concepts, such as Web services or Cloud computing where components are deployed over virtualized and dynamic environments. It has also been studied, from more a decade, in combination with testing techniques to check many aspects e.g., the security, the robustness or the conformance of different systems, such as service-oriented applications, distributed systems or object-oriented systems. In particular, conformance testing, which is the topic of this paper, aims at testing whether a black box implementation behaves correctly in relation to its specification which describes the possible interactions with the environment.

The correctness is defined by test relations which describe the confidence level between the implementation and its specification. The test verdict is then given by executing a test case set on the system under test to extract its observable behaviour. For instance, *ioco* is a well-known test relation based on trace (observable event suites) equivalence, which is often used with component based systems.

The evolution of component based engineering is still bringing new issues in testing activities. Currently, most of the component based testing approaches rely on an open environment, i.e. an environment where the component interactions inside a composition can be observed/extracted to yield a test verdict. Typically, this is a requirement for most of conformance testing approaches based on trace equivalence [BFPT06, LZCH08, iHBvdRT03, KABT10a]. Nevertheless, this assumption becomes more and more difficult to sustain especially with the recent programming techniques e.g., service or Cloud programming. Indeed, when a tester has not a sufficient access to the environment where the composition is deployed for security or technical reasons, it cannot extract the reactions of each component and cannot compute a test verdict. For instance, the dynamic nature of the Cloud architecture does not enable a tester to retrieve the messages exchanged between the services. Consequently, if the messages cannot be observed, the traces cannot be extracted and the test cannot be performed.

This paper tackles this issue by proposing a solution which recomposes the traces of a service composition under test, executed in a partially open environment like Clouds. We assume that the messages exchanged between the Web services are hidden but that we can invoke each service in a composition or at least an exact copy. We describe service compositions with the STS model (Symbolic Transition Systems [FTW05]) which is a way to describe finite automata extended with variables. STSs are not dedicated to Web service compositions, so we specialize (restrict) it with specific variables and functions to model service instances, components, and service methods called operations. We start with a test case set generated by means of a ioco based method. ioco [Tre96] is a famous test relation usually chosen for Web service testing which corresponds to trace equivalence (reaction observed) between the specification and the implemented services.

The intuition of our approach is to extract the internal messages by simulating, with a tester, a service or a client calling other service operations. So, we begin to decompose each test case to directly call the service operations with respect to the composition behaviour. More precisely, if we have a service operation which invokes another operation in a test case, thus in other terms, if we have an operation dependent on another one, we decompose the test case to test separately the two operations. By generalizing this idea, we obtain a larger test case set which produces incomplete traces, during the test execution. We reassemble them with a specific algorithm to finally produce the trace set referred in the ioco relation. With these traces, we can conclude whether the service composition conforms to the specification. We illustrate our approach with an example derived from the health record application, currently developed by the Orange Business Service company, whose purpose is to externalize patient health records with EHR (electronic heath record) which gather systematic collection of electronic health information about individual patients which can be consulted over the Internet with security restrictions.

This paper is structured as follows : Section 2 summarizes the related work concerning component based testing and gives our main motivations. In Section 3, we define the composition modelling with STSs. We also give an overview of the ioco theory. Section 4 details the test case decomposition. We provide a test case execution algorithm and show how the final traces are reassembled in Section 5. Finally, we discuss about the trace reconstruction in relation to the ioco relation in Section 6 and we conclude in Section 7.

2. Related Work and motivations

Component and composition testing have been studied by several works dealing with different areas. We describe only some of them, below.

Some works consider object-oriented systems. The authors in [WPC01, ZB07] propose methods for testing component software (object-oriented systems) from UML specifications. Like many works focusing on object-oriented systems, test cases are constructed by

means of oracles (pre, post conditions over data) representing coverage criteria which reduce the specification exploration. Gallagher and al. also propose in [GO09], technical details about an automated tool to support integration testing of object-oriented software.

Several works also consider Web services and compositions. In [GFTdlR06], the BPEL specification (Business Process Execution Language) is translated into PROMELA in order to be used by the SPIN model checking tool. In [DYZ06], the authors take BPEL specifications which are translated into Petri nets. Then, standard Petri net tools are applied to study verification, testing coverage and test case generation. In [BFPT06], the authors test the interoperability between Web services. They propose to augment the WSDL description with a UML2.0 Protocol State Machine (PSM) diagram which models the possible interactions between the service and a client. Test cases are then generated from the PSM. A framework, called the Audition framework, is defined for executing these test cases in [BP05]. In [LZCH08], the BPEL specification is translated into another model called IF, which enables the modelling of timing constraints. The test case generation is based on simulation where the exploration is guided thanks to test purposes (test requirements). A model-based black-box testing approach is described in [EGGC09]. It adapts the rule-based algorithm of the test case generation described in [GGRT06]. This algorithm consists of a simultaneous traversal of the modified execution tree and of a test purpose defined as selected finite paths of the tree.

Other works focus on components and compositions in general. Kanso and al. describe in [KABT10b] a conformance theory for components, formalized with a model which is abstract enough to subsume most state-based formalisms. The test selection is guided with test purposes. In [LLML10] Lei and al. propose a robustness testing method of stateful components modelled with State Machines (symbolic automata). The method, described in [BK09], consider component based systems with priorities where components are modelled with Labelled Transition Systems (LTSs). The authors define a technique for introducing priorities in the conformance theory. Test cases are extracted from a generated LTS representing the composition. The approach, suggested in [FRT10], aims at testing the robustness of Real-Time Component-

Based Systems. For each component, test cases are derived from two specifications : a nominal one and a degraded one. Each component is firstly tested in isolation. Then, the whole composition is robust whether the communications between components satisfy the degraded component specifications. Grieskamp and al. propose, in [GTC+05], a framework for symbolic model composition and conformance testing. Components are modelled by action machines (symbolic LTSs) which may be synchronized on the action set to produce a new specification. The conformance relation is expressed by means of conformance machines which represent the combination of a implementation (slave machine) with a specification (master machine). Then, conformance machines are explored exhaustively by means of a tool set named the XRT framework.

In most of testing methods dealing with component based systems and particularly in all the previous ones, it is assumed that all the messages corresponding to component requests are observable. This assumption makes the testing process easier to execute since the complete composition behaviour is observable. However, this assumption cannot be sustained in many cases. For instance, Cloud computing, which is a topical research subject, offers dynamic and virtualized environments where components are deployed. In such environments, messages passing between the components are hidden. This is illustrated in Figure 1. The messages depicted with black arrows are observable, whereas those depicted with red dashed arrows and modelling internal messages of the composition cannot be observed. As a general rule, as soon as a component is deployed in a environment which is partially open, most of classical testing methods cannot be applied any more.

Van der Bijl and al. proposed a different approach in [iHBvdRT03]. The specification describes the whole composition where the messages exchanged between the components are here hidden. Each component is assumed tested and working correctly though. Test cases are generated in order to check the satisfaction of the ioco test relation between the composition and the implementation under test.

Our approach offers a different viewpoint. We do not suppose that the components are tested. Indeed, in service-oriented systems, applications are often constructed with existing components owned by dif-

ferent companies and not necessarily tested. We also assume having a composition specification where all the messages, exchanged between the components, are expressed. Nevertheless, like in the Cloud computing example, we cannot observe these messages on the implementation but we can invoke each component or an exact copy.

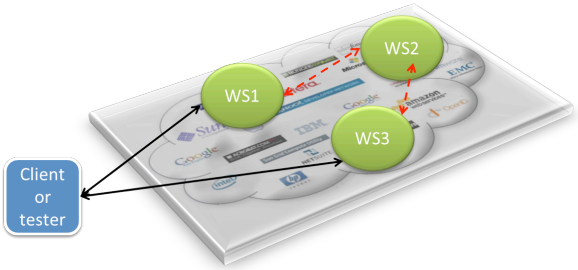

Figure 1 – Web service composition observability in a Cloud

3. Conformance testing

3.1. *Web service Composition modelling with STSs*

Several models e.g., BPEL, UML, Petri nets, process algebra, abstract state machines (ASM), have been proposed to formalize compositions of Web services, objects, etc. For instance, the BPEL language defines and manages business processes based on the interactions of Web services, called partners. A main process describes the exchanged messages and how to orchestrate them. STSs (Symbolic Transition Systems [FTW05]) have been also used with different testing methods [FTdV06, iHBvdRT03, SR10] to formalize compositions where no assumption is set on the composition style (orchestration, etc.). The STS formalism offers also a large formal background (definitions of implementation relations, test case generation algorithms, etc.). So, we based our choice on this latter to model Web service compositions and test cases. An STS is a kind of input/output automaton extended with a set of variables, with guards and assignments on variables labelled on the transitions.

Definition 3.1 *A Symbolic Transition System STS is a tuple* $< L, l_0, V, V_0, I, \Lambda, \rightarrow >$*, where :*

– L is the finite set of locations, with l_0 the initial one,

– V is the finite set of internal variables, while I is the finite set of external or interaction ones. We denote D_v the domain in which a variable v takes values. The internal variables are initialized with the assignment V_0, which is assumed to take an unique value in D_V,

– Λ is the finite set of action labels, partitioned by $\Lambda = \Lambda_I U \Lambda_O$: inputs, beginning with ?, are provided to the system, while outputs (beginning with !) are observed from it. $a(p) \in\in \Lambda \times I^n_{n>0}$ is an action where p is a finite set of external variables $p = (p_1, ..., p_k)$. We denote $type(p) = (t_1, ..., t_k)$ the type of the variable set p,

– $\rightarrow$ is the finite transition set. A transition $(l_i, l_j, a(p), \varphi, \varrho)$, from the location $l_i \in L$ to $l_j \in L$, also denoted $l_i \xrightarrow{a(p),\varphi,\varrho} l_j$ is labelled by $a(p) \in \Lambda$, $\varphi \subseteq D_V \times D_p$ is a guard which restricts the firing of the transition. Internal variables are updated with the assignment ϱ : $D_V \times D_p \rightarrow D_V$ once the transition is fired.

The STS model is not specifically dedicated (restricted) to Web service compositions. These latter are composed of Web services which may be invoked with methods called operations. This is why, for modelling, we assume that an action $a(p)$ represents either the invocation of an operation op which is denoted $opReq$ or the return of an operation op with $opResp$. For an STS $\mathcal{S}$, we denote $\mathcal{OP}(\mathcal{S}) \subseteq \Lambda$ the operation set found in $\mathcal{S}$. We also assume that the parameter set p is composed of specific variables : $c \in I$ is assigned with the called component (Web service) name and $id \in I$ represents the instance identification of a component. An instance identification helps to model a stateful service in a composition, i.e. a service which has an internal state evolving while the invocations of its operations.

We denote $component : \Lambda \times I^n_{n>0} \rightarrow D_I$ the relation which gives the component referenced into $a(p) \in \Lambda$. $\mathcal{C}(\mathcal{S}) = \{component(a_i(p)) \mid a_i(p) \in \Lambda \times I^n_{n>0}\}$ is the component set found in $\mathcal{S}$. Finally, $session : \Lambda \times I^n_{n>0} \rightarrow D_I$ is the relation which returns the instance identification for an action $a(p)$.

In the following, we assume that : STSs are deterministic, operations defined as relations are one-to-one and the dependent operations are session exclusive. These assumptions, which are explained in the remainder of the paper, are required to decompose test cases and to execute them. As a consequence, we also suppose that operations are synchronous, i.e. a Web service, invoked by an operation, returns a response immediately or does nothing. Asynchronous operations may return a response anytime, from several states and often imply indeterminism.

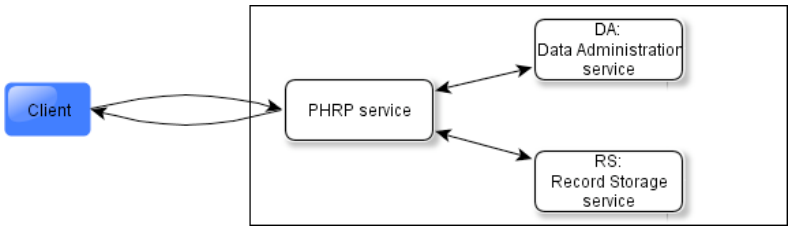

Figure 2 – A composition example

To illustrate this model, we take the example of composition given in Figure 2, derived from the health record externalization currently developed by the *Orange Business Service* company [Ora11a]. This application will aim to provide electronic health records (EHR) for both patients and practitioners. This application is currently available for sportsmen [Ora11b] but will also be available for the whole French people. The different main services linked to the specific EHR application are described below. These ones are composed of several data access layers (for access, privacy, etc.) and must respect the specific context of the directive 94/95 and the article 8 of the CNIL :

– Personal Health Record provider (PHRP) service which corresponds to the service invoked from Client applications,

– Data Administration (DA) service : Information gathering and formatting in an organization,

– Record Storage (RS) service : Storing information while respecting the anonymity systems and data protection. This service depends on others basic services e.g., patient service, medical file service, etc.,

– Workflow : Any transmission of stored information, workflows, restitution, according the level of authentication and trust imposed by health data system.

These services are currently implemented in a secured environment (Strong authentication (SSO, PKI) encryption, etc.), which is partially open and which restricts the possibilities of testing.

For sake of simplicity, we only consider, in our example, a composition of three services with few operations. The specification is given in Figure 3, with the symbol table of Figure 4. For readability reason, the Web service names are not given with the external variable c but are directly combined with operations. A patient must firstly log on the *PHRP* service, with the operation *connect*, to retrieve a key which is generated by the *DA* service. This key is mandatory to request for a health record with the *Record* operation. This one invokes the *RS* component with *patientRecord* to extract the health record and to yield it to the patient. The specification also refers to internal methods *valid* and *gen* which validate an account or a key and generate a key from an account respectively.

All the messages exchanged between the components are modelled by transitions expressing the called components, the operations and parameter variables. Each component instance is modelled by a session identification, e.g., ID1 for *PHRP*. This service is always called with the same session identifier which means that it is stateful.

An STS is also associated to an LTS (Labelled Transition System) to define its semantics. Intuitively, the semantics LTS corresponds to a valued automaton without symbolic variables : the states are labelled by internal variable values while transitions are labelled with actions and parameter values.

Definition 3.2 *The semantics of an STS* $\mathcal{S} =< L, l_0, V, V_0, I, \Lambda, \rightarrow >$ *is an LTS* $||\mathcal{S}|| =< S, s_0, \sum, \rightarrow >$ *where :*

– $L = S \times D_V$ *is the finite set of states,*

– $s_0 = (l_0, V_0)$ *is the initial state,*

– $\sum = \{(a(p), \theta) \mid a(p) \in \Lambda, \theta \in D_p\}$ *is the set of valued symbols,*

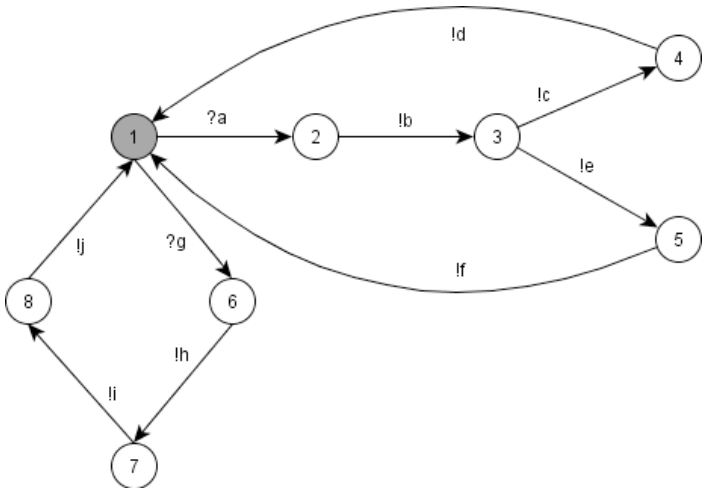

Figure 3 – An STS specification

$- \rightarrow$ is the transition relation $S \times \Sigma \times S$ deduced by the following rule :

$$\frac{l_i \xrightarrow{a(p),\varphi,\varrho} l_j, \theta \in D_p, v \in D_V, v' \in D_V, \varphi(v,\theta)\ true, v' = \rho(v,\theta)}{(l_i,v) \xrightarrow{a(p),\theta} (l_j,v')}$$

Intuitively, for an STS transition $l_i \xrightarrow{a(p),\varphi,\varrho} l_j$, we obtain an LTS one $(l_i, v) \xrightarrow{a(p),\theta} (l_j, v')$ with v an internal variable value set, if it exists a parameter value θ such that the guard $\varphi(v, \theta)$ is satisfied. Once the transition is executed, the internal variables take the value v' derived from the assignment $\varrho(v, \theta)$.

3.2. *The ioco testing theory*

For sake of simplicity, we only give the intuition of ioco testing in the paper. A complete definition can be found in [FTW05, Tre96, JÓ9]. The main idea behind the ioco conformance testing relation is to compare the observable behaviour i.e. the traces of the implementation under test with the specification ones. The absence of observation is also taken into account in the traces with a specific action, called quiescence. This

Symb	Message	Guard	Update
?a	PHRP.connectReq(String account,String id)	id==ID1	a := account
!b	DA.checkReq(String account, String id)	id==ID2 & account==a	
!c	DA.checkResp(String r,String id, String key)	r<>"invalid" & id==ID2 & valid(account) & key==gen(account)	k :=key
!d	PHRP.connectResp(String r,String key, String id)	r=="connected" & key==k & id==ID1	
!e	DA.checkResp(String r,String id, String key)	r=="invalid" & id==ID2 & ¬valid(account)	k :=null
!f	PHRP.connectResp(String r,String key, String id)	r<>"connected" & key==k & id==ID1	
?g	PHRP.RecordReq(String key, String id)	valid(k) & key==k & id==ID1	
!h	RS.patientRecordReq(String key, String id)	valid(k) & key==k & id==ID3	
!i	RS.patientRecordResp(Record rec, String id)	id==ID3	r :=rec
!j	PHRP.RecordResp(Record rec, String id)	rec==r &id==ID1	

Figure 4 – Specification symbol table

action is expressed by a new symbol δ and an augmented STS called suspension STS and denoted $\Delta(STS)$. For an STS $\mathcal{S}$, $\Delta(\mathcal{S})$ is obtained by adding a self-loop labelled by $!\delta$ for each state where quiescence may be observed. The guard of this new transition must return true for each value of $D_{V \cup I}$ which does not allow firing a transition labelled by an output. A suspension STS $\Delta(\mathcal{S})$ is associated to its suspension LTS semantics by $||\Delta(\mathcal{S})|| = \Delta(||\mathcal{S}||)$.

The set of suspension traces obtained from $\Delta(||\mathcal{S}||)$ is denoted $STraces(\mathcal{S})$. This set takes a predominant place since the ioco relation consists in comparing the non-conform traces $NCTraces$ of the specification (the complement of $STraces$) with the $STraces$ of the implementation $\mathcal{I}$:

$$\mathcal{S}\ ioco\ \mathcal{I} \Leftrightarrow STraces(\mathcal{I}) \cap NCTraces(\mathcal{S}) = \emptyset$$

To produce $STraces(\mathcal{I})$, test cases are generated from the specification $\mathcal{S}$. A test case generation algorithm for STSs can be found in [FTW05]. Test cases are defined by dedicated STSs which have a tree-like structure finished by locations labelled by *pass* or *fail*. When pass is reached during the test execution, the test case has been experimented on the implementation with success. Fail is reached otherwise. With de-

terministic specifications, when test cases are composed of several pass locations, it is required to split or to execute them several times for each pass location. For sake of simplicity, we consider in the paper that test cases are finished by one pass location only.

A test case example, derived from the STS of Figure 3 is given in Figures 5 and 6. This one checks that we can log on with the account "name,pass" and retrieve the corresponding electronic health record. For readability reason, we have not labelled the transitions to fail. These ones represent either the receipt of any other message with complementation of the guards or the observation of quiescence.

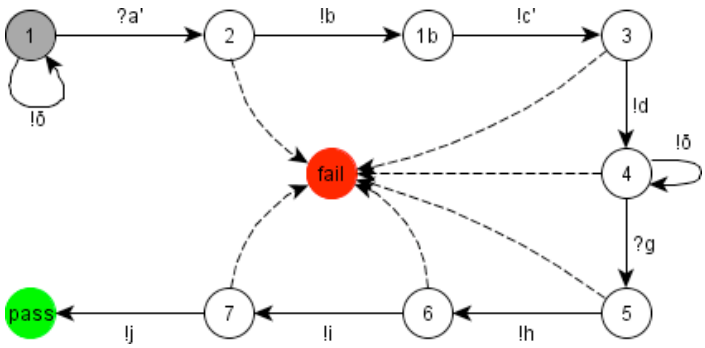

Figure 5 – A test case example

Symb	Message	Guard	Update
?a'	PHRP.connectReq(String account,String id)	account=="name,pass"&id==ID1	a :=account
!b'	DA.checkReq(String account,String id)	id==ID2 & account=="name,pass"	
!c'	DA.checkResp(String r,String id,String key)	r<>"invalid" & id==ID2 & valid(account) & key=="1234"	k := "1234"
!d'	PHRP.connectResp(String r,String key, String id)	r=="connected" & key=="1234" & id==ID1	
?g'	PHRP.RecordReq(String key,String id)	valid(k) & key=="1234" & id==ID1	
!h'	RS.patientRecordReq(String key,String id)	valid(k) & key=="1234" & id==ID3	

Figure 6 – Test case symbol table

The relation between ioco and the test suite TS is given by the soundness or the exhaustiveness of TS :

$$TS \text{ is sound} : \forall \mathcal{I}, (\mathcal{I}\ ioco\ \mathcal{S}) \Rightarrow \forall TC \in TS, \neg(TC\ mayfail\ \mathcal{I})$$

$$TS \text{ is exhaustive} : \forall \mathcal{I}, \neg(\mathcal{I}\ ioco\ \mathcal{S}) \Rightarrow \exists TC \in TS, TC\ mayfail\ \mathcal{I}$$

with TC mayfail $\mathcal{I} = \neg(\mathcal{I}$ passes $TC) \Leftrightarrow STraces(\mathcal{I}) \cap Traces_{Fail}(TC) \neq \emptyset$

A sound test suite means that it must respect the specification and that a fault cannot be found on a correct implementation. An exhaustive test suite implies that if the implementation is faulty then a fail verdict must be reached by at least one test case in TS.

The strong hypothesis behind this theory is that the implementation is supposed to behave like its reference model, in particular in term of observation. For compositions, this implies that the test architecture should be able to scan all the possible messages passing between each pair of components. So, we must have a total access to the environment where components are deployed. Having such a test architecture is difficult if the environment is partially open. On the other hand, each component (or an exact copy), taken separately, can usually be invoked by a client application, so we can observe its responses when it is called directly. But we cannot observe the requests exchanged between one component and another one.

This paper proposes a solution whose basis is to decompose each test case with respect to the operation dependency and to the Web service internal state (for those which are stateful). Then, all the services are called directly by means of a tester which simulates a component calling another one. While the test case execution, we recompose $STraces(\mathcal{I})$ to check whether the ioco relation is satisfied. In the following, we define the operation dependency and show how to decompose a test case. We show that slicing this one is not sufficient in particular when services are stateful. Then, we describe how to recompose $STraces(\mathcal{I})$ in Section 5.

4. Test case decomposition

The test case decomposition is achieved in regards to the dependency between the components participating in the composition. The intuition of the test case decomposition is to extract, from an initial test case, another test case whose purpose is to test the available operations of the composition (the operations which can be invoked directly by client applications). Then, for each of these operations, if this one also calls other operations $op_1, ..., op_n$, i.e., if this one is dependent, we extract another test case to test $op_1, ..., op_n$. This process is recursively repeated until all the dependent operations are tested by one test case.

Below, we formalize the notion of operation with relations to express and define the notion of operation dependency. Then, we describe how the test case decomposition is done.

4.1. *Operation dependency definition*

Prior to formalize the operation dependency, we briefly recall some relation properties used in the remainder of the paper.

Let $R : X \to Y$ and $S : Y \to Z$ be two relations.

– $S \circ R$ or $R; S = \{(x, z) \in X \times Y \mid \exists y \in Y, (x, y) \in R \text{ and } (y, z) \in S\}$,

– R is injective : $\forall x$ and $z \in X, \forall y \in Y$, if xRy and zRy then $x = z$,

– R is functional : $\forall x \in X, y$ and $z \in Y$, if xRy and xRz then $y = z$,

– R is one-to-one $\Leftrightarrow$ R is injective and functional.

Definition 4.1 *Let $\mathcal{S} =< L, l_0, V, V_0, I, \Lambda, \to >$ be an STS modelling a Web service composition.*

We associate the request of the operation op, modelled by a transition $l \xrightarrow{opReq(p),\varphi,\varrho} l' \in \to$, with the relation $opReq : D_{I \cup V} \to D_{I \cup V}$. Similarly, a response receipt, modelled by a transition $l \xrightarrow{opResp(p),\varphi,\varrho} l' \in \to$ is associated with the relation $opResp : D_{I \cup V} \to D_{I \cup V}$.

For a transition $l \xrightarrow{\delta,\varphi,\varrho} l' \in\!\!\rightarrow$ *expressing quiescence, we consider the reflexive relation* $\delta : D_V \rightarrow D_V$ *such that* $\forall x \in D_V, x\delta x$.

An STS path $l_i \xrightarrow{a(p),\varphi,\varrho} l_{i+1}...l_{j-1} \xrightarrow{a(p)',\varphi',\varrho'} l_j \in\!\!\rightarrow$, *is denoted* $l_i \xrightarrow{\sigma=a(p);...;a(p)'} l_{j+1}$. *We extend this notation with the notion of operation invocation. The invocation of an operation* $op \in \mathcal{OP}(\mathcal{S})$, *denoted* $l_i \overset{op}{\Rightarrow} l_j$, *refers to a path* $l_i \xrightarrow{opReq(p),\varphi,\varrho} l_{i+1}...l_{j-1} \xrightarrow{opResp(p'),\varphi',\varrho'} l_j$. *If* op *does not provide any response* $l_i \overset{op}{\Rightarrow} l_{i+1}$ *corresponds to the transition* $l_i \xrightarrow{opReq(p),\varphi,\varrho} l_{i+1}$.

Now, we can define the operation dependency which expresses that an operation is dependent if it requires to call other operations to give a result.

Definition 4.2 *Let* $\mathcal{S}$ *be an STS. An operation* $op \in \mathcal{OP}(\mathcal{S})$ *is said dependent on a relation* R, *denoted* $op \leftarrow R = opReq; R; opResp$ *if it exists the path* $l_i \xrightarrow{opReq(p)\ R\ opResp(p')} l_j \in\!\!\rightarrow$ *such that* $R \neq \delta$ *and* $R \neq \emptyset$.

Otherwise op *is said independent.*

We also denote $op \nleftarrow R$ *if* op *is independent or if* $\forall S$ *such that* $op \leftarrow S, S \neq R$.

An operation may be dependent on a list of other operations which may be independent or dependent on other ones. The specification of Figure 3 exhibits several operation dependencies. For instance, the operation *PHRP.connect* depends on *DA.check* and *PHRP.Record* depends on *RS.patientRecord*. In fact, we may reach different levels of dependency, which can be seen as a king of operation hierarchy. We express this hierarchy with a factor called the *dependency degree*.

Definition 4.3 *Let* $\mathcal{S}$ *be an STS and* $op_1 \in \mathcal{OP}(\mathcal{S})$ *be an operation.*

$\forall op_i \neq op_1 \in \mathcal{OP}(\mathcal{S})$, *if* $op_i \nleftarrow op_1$ *then we set the dependency degree of* op_1 *to 1 that we denote* op_1^{d1}. *If* $op_1 \leftarrow R$ *and* op_1^{di} *then the dependency degree of* R *is* $i+1$, *and we write* $op_1^{di} \leftarrow R^{di+1}$. *If* $op_1^{di} \leftarrow op_2^{di+1} ... \leftarrow op_n^{dk}$ *then* $op_1^{di} \leftarrow op_n^{dk}$ *with* $k > i$.

With the previous specification example of Figure 3, we can now write the operation dependencies with $PHRP.connect^{d1} \leftarrow DA.check^{d2}$ and $PHRP.Record^{d1} \leftarrow RS.patientRecord^{d2}$.

4.2. *Test case extraction*

The dependency organization leads to the test case decomposition. Intuitively, from one initial test case TC, we construct a first test case TC^1 which aims to test all the operation invocations $l_i \stackrel{op}{\Rightarrow} l_j$ in TC with op^{d1}. Then, for each invocation $l_i \stackrel{op}{\Rightarrow} l_j$ such that $\exists R, op \leftarrow R$, we construct a new test case $TC^2(l_i \stackrel{op}{\Rightarrow} l_j)$ to invoke directly the operations in R having a dependency degree equal to 2. We repeat the process until each dependent operation has its own decomposed test case.

However, the extraction of the decomposed test cases from operation invocations raises a new issue : the services may be stateful, so we need to extract paths, from the initial test case TC, which aim to call service instances from their initial states. So, let $TC =< L_{TC}, l0_{TC}, V_{TC}, V0_{TC}, I_{TC}, \Lambda_{TC}, \rightarrow_{TC} >$ be a test case. We want to extract a decomposed test case from TC to experiment an operation invocation set $\{l_i \stackrel{op_1}{\Longrightarrow} l_j, ..., l_l \stackrel{op_k}{\Longrightarrow} l_m\}$ with $ID = \{session(op_i)_{1 \leq i \leq k} \mid l_i \stackrel{op_1}{\Longrightarrow} l_j, ..., l_l \stackrel{op_k}{\Longrightarrow} l_m \in \rightarrow_{TC}\}$. We perform this extraction by means of the STS operation $keep\ ID$ in TC which aims to keep the transitions of TC labelled by actions having a session identification in ID. The result is a test case $\mathcal{P}$, composed of transitions of TC, which aims to call service instances in ID only. The operation $keep$ is defined as follow : if $ID \subseteq \{session(a(p)) \mid a(p) \in \Lambda_{TC}\}$, $keep\ ID$ in $TC =_{def} \mathcal{P} =< L_{TC}, l0_{TC}, V_{TC}, V0_{TC}, I_{TC}, \Lambda_{\mathcal{P}}, \rightarrow_{\mathcal{P}} >$ where $\Lambda_{\mathcal{P}}$ and $\rightarrow_{\mathcal{P}}$ are defined by the following inference rules :

$$R_1 : \quad \frac{l_i \xrightarrow{a_i(p),\varphi_i,\varrho_i} l_{i'} \in \rightarrow_{TC} \wedge session(a_i(p)) \notin ID \wedge l'_i \neq \text{fail} \wedge a_i(p) \neq \delta}{l_i \xrightarrow{\tau,\emptyset,\varrho_i} l'_i \in \rightarrow_{\mathcal{P}}}$$

$$R_2 : \quad \frac{l_i \xrightarrow{a_i(p),\varphi_i,\varrho_i} l_{i'} \in \rightarrow_{TC} \wedge a_i(p) = op_i Req(p) \wedge (session(a_i(p)) \in ID \vee l'_i = fail)}{l_i \xrightarrow{\delta} l_i \in \rightarrow_{\mathcal{P}} \wedge l_i \xrightarrow{?a_i(p),\varphi_i,\varrho_i} l'_i \in \rightarrow_{\mathcal{P}}}$$

$$R_3 : \quad \frac{l_i \xrightarrow{a_i(p),\varphi_i,\varrho_i} l_{i'} \in \rightarrow_{TC} \wedge a_i(p) \neq op_i Req(p) \wedge (session(a_i(p)) \in ID \vee l'_i = fail)}{l_i \xrightarrow{!a_i(p),\varphi_i,\varrho_i} l'_i \in \rightarrow_{\mathcal{P}}}$$

These rules produce an STS $\mathcal{P}$ which has the same structure as the original test case TC. However, these rules hide, with the τ symbol, the actions of TC which express the call of component instances not referenced in ID. So, during the test execution, this will help to recompose the final traces. More precisely, the rule R_1 aims to hide actions of TC which do not have a session identification in ID. Transitions leading to fail or labelled by δ are not hidden. These ones do not interfere with the call of components. R_2 and R_3 mean that if the test case transition has a session identification in ID then this one must be kept. R_2 modifies the action set of the initial test case since all the operation requests, which are not hidden, are translated into inputs. These inputs will be executed by a tester during the test execution to call each instance of Web services in ID. In R_2 when operation requests are translated into inputs, we also add a self-loop transition labelled by δ to express quiescence.

Now, let op_1 and op_2 be two operations such that $op_1^{di} \leftarrow op_2^{dk}$. To test op_2, the previous rules will extract, from a test case TC, all the transitions $l_j \xrightarrow{a_j(p),\varphi,\varrho} l_{j+1}$ which have the same instance identification $session(op_2)$. If we suppose that the two operations are called in the same instance, $session(op_1) = session(op_2)$, we obtain a test case part $l_i \xrightarrow{op_1 Req(p),\varphi,\varrho} l_{i+1} ... l_j \xrightarrow{op_2 Req(p'),\varphi',\varrho'} l_{j+1}$. This one means that we invoke successively op_1 and op_2. However the operation op_1 implicitly invokes op_2 since $op_1^{di} \leftarrow op_2^{dk}$. Thus, we obtain a case of dependency that we cannot break. So, in the remainder of the paper, we set the following assumption which means that if an operation op_i is dependent on op_j then the two operations must not belong to the same component instance.

Operation session exclusion : Let $\mathcal{S}$ be an STS, and $(op_i, op_j) \in \mathcal{OP}(\mathcal{S})^2$ be two operations. If $op_i^{dm} \leftarrow op_j^{dn}$ then $session(op_i) \neq session(op_j)$.

Now, we are ready for the test case decomposition algorithm. Let $TS = \{TC_1, ..., TC_m\}$ be the initial test suite. Each test case $TC_n \in TS$ is decomposed with Algorithm 1. The latter produces $TS' = \{TC'_1, ..., TC'_m\}$ where $TC'_n \in TS'$ corresponds to the decomposition of TC_n. From an initial test case $TC_n \in TS$, the algorithm begins to construct a session identification set from operations of TC_{pass} having a dependency degree equal to 1 (lines 2-3). Then, we call the TGGen procedure (line 5) which extracts a decomposed test case $TC_n^d(l_i \overset{op}{\Rightarrow} l_j)$ with the $keep$ operation (line 9). We repeat the process for each operation invocation $l_k \overset{op'}{\Rightarrow} l_l$ found in the path p such that the dependency degree of op' is equal to d. So, the algorithm constructs a session identification set from operations in $l_k \overset{op'}{\Rightarrow} l_l$ having a dependency degree equal to $d + 1$ (line 13). TCGen is recursively called (line 14) to produce $TC_n^{d+1}(l_k \overset{op'}{\Rightarrow} l_l)$. The algorithm ends when each invocation $l_i \overset{op}{\Rightarrow} l_j$, with op a dependent operation, has its own test case $TC_n^d(l_i \overset{op}{\Rightarrow} l_j)$. So, the test case $TC_n \in TS$ is decomposed by $TC'_n = \{TC_n^1(\emptyset)\} \cup \{TC_n^d(l_i \overset{op}{\Rightarrow} l_j) \mid d > 1, op \text{ is a dependent operation}\}$.

From the initial test case given in Figure 5, we obtain the test case set TC' illustrated in Figure 7. The first test case $TC^1(\emptyset)$ gathers the operations having a dependency degree equal to 1 (***PHRP.connect, PHRP.Record***). The second test case $TC^2(l_1 \overset{PHRP.connect}{\Longrightarrow} l_4)$ invokes directly the operation *DA.check*. The last test case $TC^3(l_4 \overset{PHRP.Record}{\Longrightarrow} l_8)$ aims to invoke the operation *RS.patientRecord*. The order of the actions is kept in comparison with the initial test case. Thus, when these decomposed test cases will be executed, we will obtain incomplete traces which will be overlaid to produce final ones. The request of the operations *DA.check* and *RS.patientRecord* which were expressed by output actions in the specification and in the initial test case, are now modelled with inputs thanks to the *keep* operation (rule R_2). This modification means that the operations will be invoked directly by the

tester. Since these ones are translated into inputs, self-loop transitions labelled by δ are added to express quiescence.

input : A test suite TS
output: A test suite TS'

foreach *test case* $TC_n \in TS$ **do**
  TC_{pass} is the acyclic path of TC_n finished by pass;
  $ID = \{session(op) \mid l_i \stackrel{op}{\Rightarrow} l_j \in TC_{pass}, op^{d1}\}$;
  $TC'_n := \emptyset$;
  TCGen($TC_{pass}, ID, \emptyset, 1$);
  $TS' := TS' \cup \{TC'_n\}$;
end

TCGen(STS path p, instance identification set ID, operation invocation $l_i \stackrel{op}{\Rightarrow} l_i$, degree d)
//extract a decomposed test case with the $keep$ operation;
$TC^d_n(l_i \stackrel{op}{\Rightarrow} l_j)$:= $keep\ ID$ in TC_n;
$TC'_n := TC'_n \cup TC^d_n(l_i \stackrel{op}{\Rightarrow} l_j)$;
foreach *operation invocation* $l_k \stackrel{op'}{\Rightarrow} l_l \in p$ *with* op'^{dd} **do**
  if op' *is a dependent operation* **then**
    $ID = \{session(op) \mid l_q \stackrel{op}{\Rightarrow} l_r \in l_k \stackrel{op'}{\Rightarrow} l_l, op^{dd+1}\}$;
    TCGen($l_k \stackrel{op'}{\Rightarrow} l_l$,$ID$, $l_k \stackrel{op'}{\Rightarrow} l_l, d+1$);
  end
end

Algorithm 1: Test suite decomposition

5. The ioco satisfiability with decomposed test cases

The previous section showed how to decompose an initial test case according to the operation dependency. However, each decomposed test case will provide a partial trace only, while testing. These partial traces are not sufficient to conclude whether the implementation is ioco-conforming to its specification. This section presents how to recompose these partial traces to recover complete traces. Then, we show that the resulting traces can be used to check the ioco relation satisfiability.

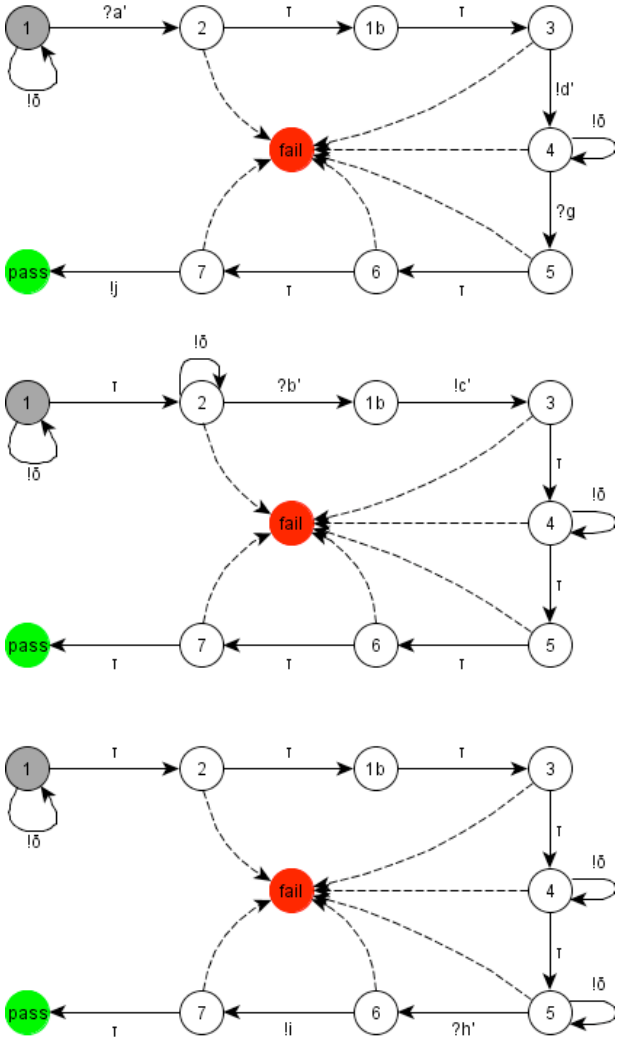

Figure 7 – A decomposed test case

5.1. *Test execution and trace recomposition*

To reason about conformance, we assume that an implementation under test $\mathcal{I}$ is modelled by a suspension LTS and should behave as its specification ($\Delta(||\mathcal{S}||)$). During the test execution, the tester aims to recompose $STraces(\mathcal{I})$ by executing the test suite $TS' = \{TC'_1, ..., TC'_m\}$ where TC'_n is a decomposed test case.

The trace recomposition is done in referring to the operation dependency. Nevertheless, the dependency between operations cannot be explicitly observed since the messages exchanged between the components are not observable. However, if we assume that any dependent operation of the specification is one-to-one, we can affirm that an operation *op is probable dependent on* a relation R, if *op* responds correctly during the test.

Definition 5.1 *Let TC be a test case composed of the invocation $l_i \stackrel{op}{\Rightarrow} l_j \in \rightarrow_{TC}$ such that $\exists R, op \leftarrow R$. If op is one-to-one and if the implementation under test $\mathcal{I}$ passes TC then op is said probable dependent on R.*

The relation R is a composition over $\mathcal{OP}(\mathcal{S}) \cup \{\delta\}$. Each relation in $\mathcal{OP}(\mathcal{S}) \cup \{\delta\}$ is one-to-one, so R is one-to-one. $op \leftarrow R$ corresponds to $opReq; R; opResp$. So, Let $w, x, y, z \in D_{V \cup I}$ such that w opReq x, xRy and y opResp z. $\mathcal{I}$ passes TC implies that the operation op has given a correct response. So, We suppose that we call $opReq(w)$ and that we observe $opResp(z)$. $opReq$ is functional thus it exists an unique $x' \in D_{V \cup I}$ such that w opReq x'. $opResp$ is injective thus it exists an unique $y' \in D_{V \cup I}$ such that y' opResp z. Thereby, op sounds to be dependent on the relation R such that $x'Ry'$ with $x = x'$ and $y = y'$.

The trace recomposition, performed by a tester, is described in Algorithm 2. For a decomposed test case $TC'_n = \{TC^1_n(\emptyset)\} \cup \{TC^d_n(l_i \stackrel{op}{\Rightarrow} l_j) \mid d > 1, op \text{ is a dependent operation}\}$, the tester tries to assemble a trace $trace_n(\mathcal{I})$ with the *Exec* procedure (line 3). It begins to test $TC^1_n(\emptyset)$ on $\mathcal{I}$. For instance, if this one is composed of the invocation $l_i \stackrel{op}{\Rightarrow} l_j = l_i \xrightarrow{?opReq(p),\varphi,\varrho} l_{i+1} \stackrel{\tau}{\rightarrow} l_{i+2}...l_{j-2} \stackrel{\tau}{\rightarrow} l_{j-1} \xrightarrow{!opResp(p),\varphi',\varrho'} l_j$, the tester invokes the operation op, then ignores the actions τ and waits

for a response or observes quiescence (the lack of response). The resulting actions are injected into $trace_n(\mathcal{I})$ (lines 6-8). For each test case invocation $l_k \overset{op'}{\Rightarrow} l_l \in TC_n^d(l_i \overset{op}{\Rightarrow} l_j)$, if $\mathcal{I}$ passes $TC_n^d(l_i \overset{op}{\Rightarrow} l_j)$, then the operation op' has responded correctly. According to Definition 5.1, if $op' \leftarrow R$, op' sounds probable dependent on R. Thereby, the tester executes the test case $TC_n^{d+1}(l_k \overset{op'}{\Rightarrow} l_l)$ to test and to retrieve the traces of the operations invoked by op' (lines 10-11). The process continues for each dependent operation to finally recompose $trace_n(\mathcal{I})$. If an operation does not return a correct response then the trace reconstruction is stopped (line 13).

input : A test suite TS'
output: $STraces(\mathcal{I})$

foreach *test case* $TC'_n \in TS'$ **do**
  $trace_n(\mathcal{I}) = t_0...t_k$ is a trace;
  Exec($TC_n^1(\emptyset)$, $trace_n(\mathcal{I})$);
  $STraces(\mathcal{I}) := STraces(\mathcal{I}) \cup trace_n(\mathcal{I})$;
end

Exec(test case $TC_n^d(l_i \overset{op}{\Rightarrow} l_j)$, trace $trace_n(\mathcal{I})$)
$trace'(\mathcal{I}) = Test(TC_n^d(l_i \overset{op}{\Rightarrow} l_j))$;
foreach $t'_k \in trace'(\mathcal{I}) \neq \tau$, $i < k < j-1$ **do**
  $t_k := t'_k$;
end

foreach *invocation* $l_k \overset{op'}{\Rightarrow} l_l \in TC_n^d(l_i \overset{op}{\Rightarrow} l_j)$ *with* op' *a dependent operation* **do**
  if $\mathcal{I}$ *passes* $TC_n^d(l_i \overset{op}{\Rightarrow} l_j)$ *(*op' *is probable dependent)* **then**
    Exec($TC_n^{d+1}(l_k \overset{op'}{\Rightarrow} l_l)$, $trace_n(\mathcal{I})$);
  end
  else
    Stop;
  end
end

Algorithm 2: Tester algorithm

5.2. *Ioco satisfiability*

Let $\mathcal{S}$ be an STS and $\mathcal{I}$ an implementation under test. We suppose that we have $\mathcal{I}$ ioco $\mathcal{S}$ and a test case TC which produces the trace $t_1...t_n$ with a classical ioco-oriented testing method. If we apply Algorithms 1 and 2 on TC and $\mathcal{I}$, we obtain the same trace $t_1...t_n$. The sketch of proof is the following :

TC is transformed by Algorithm 1 into $\{TC^1(\emptyset)\} \cup \{TC^d(l_i \stackrel{op}{\Rightarrow} l_j) \mid op$ is dependent, $d > 1\}$. $TC^1(\emptyset) = p.l_1 \stackrel{op_1}{\Rightarrow} l_2...l_i \stackrel{op_i}{\Rightarrow} l_j...l_k \stackrel{op_k}{\Rightarrow} l_n$ with op_i^{d1} $(1 \leq i \leq k)$ (Algorithm 1 and the $keep$ operation). p is a path which calls services in $\{component(op_i) \mid 1 \leq i \leq k\}$ from their initial states. Since the dependency degree of each operation invocation is equal to 1, $p = \emptyset$ here. Consider the invocation $l_i \stackrel{op_i}{\Rightarrow} l_j$ with $c = component(op_i)$ the service which is called :

– if op_i is not dependent, $l_i \stackrel{op_i}{\Rightarrow} l_j$ corresponds to one of the following $TC^1(\emptyset)$ paths : $l_i \xrightarrow{op_i Req} l_{i+1}$, $l_i \xrightarrow{\delta} l_{i+1}$, $l_i \xrightarrow{op_i Req\ op_i Resp} l_{i+2}$ or $l_i \xrightarrow{op_i Req \delta ... \delta op_i Resp} l_j$ (We have supposed that operations are synchronous and that STSs are deterministic). From these cases, Algorithm 2, produces respectively t'_i, δ, $t'_i t'_{i+1}$ or $t'_i \delta ... \delta t_{j-1}$ with $t'_i = t_i$, $t'_{i+1} = t_{i+1}$ and $t'_{j-1} = t_{j-1}$ since the service c is called from its initial state and since the $keep$ operation keeps the order of the initial test case transitions. So, we obtain the same observable events as those produced by a ioco based method,

– if op_i is dependent, $l_i \stackrel{op_i}{\Rightarrow} l_j$ corresponds to $l_i \xrightarrow{op_i Req} l_{i+1} \xrightarrow{\tau} l_{i+2}...l_{j-2} \xrightarrow{\tau} l_{j-1} \xrightarrow{op_i Resp} l_j$ ($keep$ operation). Algorithm 2 produces the partial trace $t'_i \tau ... \tau t'_{j-1}$ with $t'_i = t_i$ and $t'_{j-1} = t_{j-1}$ since the service c is called from its initial state and since the $keep$ operation keeps the order of the initial test case transitions. Since we suppose that $\mathcal{I}$ ioco $\mathcal{S}$, $\mathcal{I}$ passes $TC^1(\emptyset)$ and op_i is probable dependent (Algorithm 2 line 10). Thereby, it exists a decomposed test case $TC^2(l_i \stackrel{op_i}{\Rightarrow} l_j)$ (Algorithm 1 lines 12-14) such that $TC^2(l_i \stackrel{op_i}{\Rightarrow} l_j) = p'.l_{i+1} \xrightarrow{op_{i+1} Req} l_k...l_o \stackrel{op_o}{\Rightarrow} l_p...l_q \stackrel{op_q}{\Rightarrow} l_{j-1}$, with p' a path which aims to call services in $\{component(op_o) \mid i+1 \leq o \leq q\}$ from their initial states.

We can recursively apply the same reasoning on $TC^2(l_i \stackrel{op_i}{\Longrightarrow} l_j)$ and on each dependent and independent operation until there is no operation to call. Since the $keep$ operation produces paths with the same transition order as the initial test case and since the decomposed test cases always call components from their initial states, we finally obtain a ioco compatible trace $t_1...t_n$.

If we suppose now that $\neg(\mathcal{I}$ ioco $\mathcal{S})$, we may obtain a trace $t_1...t_n$ either composed of incorrect responses or of τ symbols with Algorithm 2. For both cases, the application of $t_1...t_n$ on the initial test case TC leads to a fail location. In other terms, $t_1...t_n \in Traces_{FAIL}(TC)$.

So, let $\mathcal{S}$ be an STS, $\mathcal{I}$ an implementation under test and $TC \in TS$ a test case :

$\mathcal{I}$ ioco $\mathcal{S} \Rightarrow$ the application of Algorithms 1 and 2 on TC produces a trace $t \in STraces(\mathcal{I})$ such that $t \notin Traces_{FAIL}(TC)$

$\neg(\mathcal{I}$ ioco $\mathcal{M}) \Rightarrow$ the application of Algorithms 1 and 2 on TC produces a trace$t \in STraces(\mathcal{I})$ such that $t \in Traces_{FAIL}(TC)$

6. Discussion

Both the test case decomposition and the test execution described above cost time. The time complexity remains reasonable though, since it is proportional to $\mathcal{O}((k+1)n^2)$ with k the number of test cases and n the highest transition number in a test case. For comparison purposes, executing test cases on an architecture where all the messages can be observed is $\mathcal{O}(kn)$. From an initial test case TC of n transitions, we have at most n operation invocations. In the worst case, we construct one test case per invocation by covering TC two times (Algorithm 1). So, for k test cases, the time complexity is $\mathcal{O}(kn^2)$. We have at most $(n+1)$ test cases generated from TC whose length cannot exceed n transitions. So, the test execution is proportional to $\mathcal{O}(n^2)$.

This time complexity may be reduced by considering the nature of the tested system. Indeed, we consider services which could be called separately in parallel by several testers. For a decomposed test case $TC'_n = \{TC^1_n(\emptyset)\} \cup \{TC^d_n(l_i \stackrel{op}{\Rightarrow} l_j) \mid d > 1, op$ is a dependent operation$\}$, each test case TC^d_n calls instances of

Web services from their initial states. So, a tester could execute each TC_n^d in parallel by using its own Web service instances, in condition that there is one unique instance per Client application (no shared session). The test execution is then proportional to $\mathcal{O}(n)$.

The second point of attention concerns the composition of $STraces(\mathcal{I})$ while testing. Once we have $STraces(\mathcal{I})$, we check for any test case TC whether $STraces(\mathcal{I}) \cap Traces_{Fail}(TC) \neq \emptyset$ and finally we conclude whether $\mathcal{S}$ ioco $\mathcal{I}$. This raises the following question : should we still name the test relation ioco ? Test cases are generated by ioco based methods and we reconstruct complete traces in $STraces(\mathcal{I})$. Nevertheless, we recompose traces by supposing that operations are probable dependent whereas this property does not exist in the ioco theory. So, is it required to redefine a weakest relation ? We do not think so for the following reason : if we have one-to-one operations and $op \leftarrow R$ in the specification, and if the implementation of op responds correctly during the test, we are faced with the following choices : 1) op is independent but simulates the relation R and gives the correct response, 2) op is dependent on other operations $op \leftarrow R'$ which have exactly the same behaviour as R, 3) op is really dependent on R. Implementing the two first cases from a given specification sounds unlikely since they correspond to nonsense modifications of the specification. This is why we believe that only the last case is reasonable. Nevertheless, it is manifest that proposing a test relation based on the trace composition would be still interesting.

7. Conclusion

This paper proposes a method for testing Web service compositions deployed inside a partially open environment where the messages passing between the components are hidden. This method tests the composition behaviour and does not assume that each component is tested and conform. To retrieve all the traces of the composition, the test case set is firstly decomposed over the operation dependency degree which models the interleaving between the Web services of the composition. By assuming that each of them (or an exact copy) can be invoked directly by a client application, the tester executes the decomposed test cases,

retrieves local traces from services and recomposes them to produce the suspension traces $STraces$. Then, it becomes possible to check whether the implementation is ioco-conforming to its specification. We are currently implementing the method. We have experimented it with success on two available services of the health record application. From an initial set of twenty test cases formulated manually, the test case decomposition algorithm has generated roughly fifty decomposed test cases. These ones have been executed by a second preliminary tool which implements the tester algorithm (Algorithm 2). To execute the decomposed test cases, we have manually translated them into the SOAPUI format. Then, the tester uses the SOAPUI tool [Evi11] whose purpose is to experiment Web services with unit test cases.

An immediate line of future work is to reduce the time complexity of the method either by proposing optimized algorithms or by proposing pragmatic solutions such as testing components in parallel. We have also set several assumptions on the specification, in the paper. These ones are required to decompose and to execute a test suite. For instance, we suppose that specifications are deterministic. However, nondeterministic specifications may be taken into account, in a future work, with some restrictions [JMR06]. As a consequence, asynchronous operations could be tested too. Other assumptions sound more difficult to remove. For instance, an operation cannot be said probable dependent on another one, if it is not one-to-one (Definition 5.1).

This notion of probable dependency is the weak part of the test execution and need to be investigated in future works. It would be very interesting to guarantee that an operation is dependent and not only probable dependent. Nevertheless, we cannot observe the internal messages of the composition since these latter are assumed hidden. A trivial solution could be proposed by implementing in each service a king of historical log which could be easily available. Nevertheless, this kind of technical solution cannot be generalized and is useless if we wish to use existing components. A formal solution would be more relevant.

Références

[BFPT06] A. Bertolino, L. Frantzen, A. Polini, and J. Tretmans. Audition of web services for testing conformance to open specified protocols. In R. Reussner, J. Stafford, and C. Szyperski, editors, *Architecting Systems with Trustworthy Components*, number 3938 in LNCS, pages 1–25. Springer-Verlag, 2006.

[BK09] I. B. Bourdonov and A. S. Kossatchev. Systems with priorities : Conformance, testing, and composition. *Program. Comput. Softw.*, 35 :198–211, July 2009.

[BP05] Antonia Bertolino and Andrea Polini. The audition framework for testing web services interoperability. In *31st EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 134–142, 2005.

[DYZ06] Wen-Li Dong, Hang Yu, and Yu-Bing Zhang. Testing bpel-based web service composition using high-level petri nets. In *Proceedings of the 10th IEEE International Enterprise Distributed Object Computing Conference*, pages 441–444, Washington, DC, USA, 2006. IEEE Computer Society.

[EGGC09] Jose Pablo Escobedo, Christophe Gaston, Pascale Gall, and Ana Cavalli. Observability and controllability issues in conformance testing of web service compositions. In *TESTCOM '09/FATES '09, 21st IFIP WG 6.1 International Conference on Testing of Software and Communication Systems*, pages 217–222, Berlin, Heidelberg, 2009. Springer-Verlag.

[Evi11] Eviware. Soapui. 2011. http ://www.soapui.org/.

[FRT10] Hacène Fouchal, Antoine Rollet, and Abbas Tarhini. Robustness testing of composed real-time systems. *J. Comp. Methods in Sci. and Eng.*, 10 :135–148, September 2010.

[FTdV06] Lars Frantzen, Jan Tretmans, and René de Vries. Towards model-based testing of web services. In Antonia Bertolino and Andrea Polini, editors, *in Proceedings of International Workshop on Web Services Modeling and Testing (WS-MaTe2006)*, pages 67–82, Palermo, Sicily, ITALY, June 9th 2006.

[FTW05] L. Frantzen, J. Tretmans, and T.A.C. Willemse. Test Generation Based on Symbolic Specifications. In J. Grabowski and B. Nielsen, editors, *Formal Approaches to Software Testing – FATES 2004*, number 3395 in Lecture Notes in Computer Science, pages 1–15. Springer, 2005.

[GFTdlR06] José García-Fanjul, Javier Tuya, and Claudio de la Riva. Generating test cases specifications for compositions of web services. In Antonia Bertolino and Andrea Polini, editors, *in Proceedings of International Workshop on Web Services Modeling and Testing (WS-MaTe06)*, pages 83–94, Palermo, Sicily, ITALY, June 2006.

[GGRT06] Christophe Gaston, Pascale Le Gall, Nicolas Rapin, and Assia Touil. Symbolic execution techniques for test purpose definition. In *TestCom*, pages 1–18, 2006.

[GO09] Leonard Gallagher and Jeff Offutt. Test sequence generation for integration testing of component software. *Comput. J.*, 52(5) :514–529, 2009.

[GTC+05] Wolfgang Grieskamp, Nikolai Tillmann, Colin Campbell, Wolfram Schulte, and Margus Veanes. Action machines - towards a framework for model composition, exploration and conformance testing based on symbolic computation. In *Proceedings of the Fifth International Conference on Quality Software*, QSIC '05, pages 72–82, Washington, DC, USA, 2005. IEEE Computer Society.

[iHBvdRT03] ir. H.M. Bijl van der, Dr.ir. A. Rensink, and Dr.ir. G.J. Tretmans. Component based testing with ioco., 2003.

[JÓ9] Thierry Jéron. Symbolic model-based test selection. *Electron. Notes Theor. Comput. Sci.*, 240 :167–184, July 2009.

[JMR06] T. Jéron, H. Marchand, and V. Rusu. Symbolic determinisation of extended automata. In Springer Science and Business Media, editors, *4th IFIP International Conference on Theoretical Computer Science,Santiago, Chile*, volume 209/2006, pages 197–212, 2006.

[KABT10a] Bilal Kanso, Marc Aiguier, Frédéric Boulanger, and Assia Touil. Testing of abstract components. In *Proceedings of the 7th International colloquium conference on Theoretical aspects of computing*, ICTAC'10, pages 184–198, Berlin, Heidelberg, 2010. Springer-Verlag.

[KABT10b] Bilal Kanso, Marc Aiguier, Frédéric Boulanger, and Assia Touil. Testing of abstract components. In *Proceedings of the 7th International colloquium conference on Theoretical aspects of computing*, ICTAC'10, pages 184–198, Berlin, Heidelberg, 2010. Springer-Verlag.

[LLML10] Bin Lei, Zhiming Liu, Charles Morisset, and Xuandong Li. State based robustness testing for components. *Electron. Notes Theor. Comput. Sci.*, 260 :173–188, January 2010.

[LZCH08] M. Lallali, F. Zaidi, A. Cavalli, and I. Hwang. Automatic timed test case generation for web services composition. In IEEE Computer Society Press, editor, *The 6th IEEE European Conference on Web Services (ECOWS'08)*, pages 53–63, Dublin, november 2008.

[Ora11a] Orange. Orange healthcare system. 2011. http ://www.orange.com/en_EN/group/activities_key /health/.

[Ora11b] Orange Business Service (Almerys). Sportmen health record management. 2011. http ://www.orange-

business.com/fr/presse/communiques/offres/suivi-medical-en-ligne-des-sportifs.html.

[SR10] Sébastien Salva and Issam Rabhi. Stateful web service robustness. In *ICIW '10 : Proceedings of the 2010 Fifth International Conference on Internet and Web Applications and Services*, pages 167–173, Washington, DC, USA, 2010. IEEE Computer Society.

[Tre96] Jan Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software - Concepts and Tools*, 17(3) :103–120, 1996.

[WPC01] Ye Wu, Dai Pan, and Mei-Hwa Chen. Techniques for testing component-based software. In *Proceedings of the Seventh International Conference on Engineering of Complex Computer Systems*, pages 222–232, Washington, DC, USA, 2001. IEEE Computer Society.

[ZB07] Weiqun Zheng and Gary Bundell. Model-based software component testing : A uml-based approach. *ACIS International Conference on Computer and Information Science*, 0 :891–899, 2007.