ParTes: A Test Generation Strategy for Choreography Participants

Francesco de Angelis, Daniele Fanì, Andrea Polini

ParTes: A Test Generation Strategy for Choreography Participants

Francesco De Angelis, Daniele Fanì
Computer Science Division – School of Science and Technology
University of Camerino
Via Madonna delle Carceri, 9 - 62032 Camerino, Italy
{francesco.deangelis, daniele.fani}@unicam.it

Andrea Polini
ISTI – CNR
Via G. Moruzzi, 1
56124 Pisa, Italy
andrea.polini@isti.cnr.it

***Abstract*—Inter-organizational business processes permit to specify how different organizations can integrate to carry on business activities together. In this context choreography specifications provide a particularly useful view permitting to define how different organizations should interact and exchange messages in order to fruitfully cooperate. Tools and mechanisms permitting to check that a stakeholder, and its provided e-services, are able to correctly cooperate according to the global specification become an important and useful asset. This is particularly true when open specifications are considered and services dynamically integrate with each other at run-time.**

This paper proposes a novel derivation strategy for test case skeletons, which can be successively refined and concretized to check the behaviour of parties willing to play a role within a choreography enactment. The very basic idea is to derive test cases from the possible interaction traces included in the choreography specification handled as a workflow graph. The selection of traces to use for test derivation purpose is driven by a specifically conceived technique to work flow refactoring which permits to reduce the number of interleavings to explore, in particular when parallel statements are considered.

***Index Terms*—Service Testing, Service Choreography, Model Based Testing**

I. Introduction

The Internet was built to support interactions among computing systems several years ago with low-level communication capabilities and few opportunities for business-oriented applications. Today the scenario is different: the use of remote software components (services) has became an indispensable part of the application logic of business-oriented applications. This scenario has been partially enabled by the Service Oriented Architecture (SOA) vision, where services can integrate and collaborate in a loosely coupled way to achieve a collective business objective. Thus a service based application is no more an isolated service but a complex integration of remote functionalities provided by different services. This integration can refer both to an inter-organizational setting, as for a B2B scenario, and intra-organizational as for services supporting internal processes of a single organization. Orchestration and choreography are two different ways to organize service integration and to specify the message traffic among participants. Within a service orchestration scenario there is a specification (possibly executable) which acts as a central point of control and it directs the interactions with the involved service partners. On the other side a choreography can be considerated merely as a message exchange protocol among different parties. A choreography does not define any directly executable specification to coordinate the interactions among the participants. It just tells a complete multi-party story so that participants can determine their role by isolating the parts in which they are involved. The overall objective of a choreography is typically the provisioning of complex inter-organizational service based applications which result from the interoperation of many services provided by different organizations. Through the participation to a choreography execution (enactment) each participant aims at reaching a "personal" objective obviously related to the mission of the organization. The choreography specification permits to mediate among these, possibly diverging, interests and enables a fruitful integration and cooperation of different parties.

The organizations interested in engaging in the composition defined by a choreography specification should respect the established rules as both in terms of business objectives (high level description of a choreography), and from a technological point of view (low level description in terms of messages exchange). Partners are typically involved in choreography enactments through provided services that are distributed both physically and temporarily, with service participants entering in and exiting from a choreography. Thus, it is easy to realize that interoperability becomes a real challenge.

Testing can be a useful technique to reduce the risk of interoperability issues. Considering that a service can participate to choreographies possibly defined after service deployment, it becomes evident that it is not possible to test a service as part of a "traditional" development process where the source code is often available. On the other side we are in urgent need of testing techniques driven by choreography specifications. As for any integration specification test cases which seem able to identify possible issues are those related to interactions spanning over many different elements (participants in our case) of the composition. While using model checking techniques to check the possible interactions can seem a good solution, it easily suffers by the state-explosion problem. Our objective has been to derive a testing strategy that, taking in input a choreography specification, it is able to derive skeleton test cases for the different participants, or subset of them, in order to check if a given service would able to play a role within a choreography enactment. The element of novelty of our approach is considering dependencies we defined among interleavings to select just traces we consider "good" to create tests, instead of generating all as a standard model-checking approach would do.

We start our paper describing in Section VI some works related to this strategy and in Section II some preliminar notions about BPMN2 and message dependencies. Section III shows the overall reduction strategy showing the main algorithm that we refine in Section IV. In Section V we complete the strategy adding testing consideration and Section VII closes the work and discuss future improvements.

II. Preliminaries

A. Choreography specification with BPMN2

The Business Process Modeling Notation (BPMN2) is the de-facto standard for specifying processes at an high level of abstraction and it can be used to provide a representation of processes within a single organization or several cooperating organizations. The notation includes diagrams to describe processes and a diagram for choreography specification in which the diagram elements describe the exchange of messages among participants and their possible flow. A *Process* diagram describes a sequence of *Activities* within an organization with the objective of describing a workflow of a single subject, while a *Choreography* diagram (and *Collaboration* diagrams in general) models the interactions among processes activated by different organizations. In this new setting *Processes* are related to *Participants* that can be involved in a choreography according also to their internal process. A *Choreography* diagram formalizes the way in which different participants coordinate the exchange of messages leading to the completion of the overall workflow, respecting to the local workflows of each individual participant. A single interaction between two participants can be described by a *ChoreographyTask* ("CT" for short). A choreography describes then the order in which the *ChoreographyTasks* can be executed. CTs can be arranged to be executed sequentially or in parallel and following conditional statements through the use of gateways.

BPMN2 allows to define interactions without specifing data related information. Conditional flows can be represented as non-deterministic choices and external notations can be used to add additional information. In this paper we provide a strategy for deriving test skeletons starting from a BPMN2 specification. Skeletons structures highlight relevant interaction sequences and can be refined to concrete tests adding details about data to be used.

B. Handling message dependencies

In testing services involved in choreography enactments we should focus on possible interaction sequences that can affect the internal state of a participant and then the global behaviour of the composition. A way to know how an interaction could change the internal state of a service is to derive the flow of messages exchanged with the other participants according to the choreography specification. Assuming that in general messages received from a sender can alter the internal behavior of the receiver, we say that there is a potential *message dependency* between them. Our approach intends to identify those interaction traces which are foreseen by the choreography specification, and that can lead to different behaviour of the involved services. In this respect, the algorithm does not explore those traces which can identify a different interleaving of the exchanged messages but nevertheless does not lead to a possible different behaviour for the involved services.

For instance in Figure 1, we consider a BPMN *ParallelGateway* with two CTs executed concurrently named a and b. Two different execution traces are possible: one in which a follows b, and the other in which b follows a. In subfigure (a) the two possible execution traces of the CTs are not equivalent because there is a potential message dependency in the participant P2 that can be altered by the messages sent by P1. We are interested in this situation to produce a suitable test skeleton which can highlight a potential issue in the involved services. Conversely, execution traces in subfigure (b) are equivalent as the participants do not affect each other. In this case we can select one of them as representative for the concurrent execution and build a single skeleton accordingly.

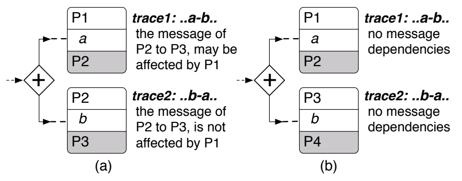


Fig. 1. Execution orders of a *ParallelGateway* (CT symbols shows the receiver participant in gray)

In the following, we will use the notation *sender(CT)* and *receiver(CT)* to indicate respectively the participant that sends/receives a message in a given CT, and $(a, b, \ldots, n)$ to describe the execution trace of CT a followed by CT b, followed by other CTs and finally by CT n. We have identified different patterns of how dependency between parallel CTs may occur and lead to possible issues which we want to test. Each pattern, taken individually, produces a different number of execution traces:

- "no dependency". This case relates to a flow in which there are no message dependencies, as shown in Figure 2(a). Here, respecting the flow which impose that CT b must be executed after CT a and that CT d after CT c, there are six possible execution traces. Nevertheless our strategy returns just one of them that becomes the representative one. The tasks order is considered not relevant since the CTs do not affect each other.
- "receiver-sender" pattern. This case relates to a flow in which $receiver(CT) = sender(CT')$ with CT and CT' belonging to different branches of a parallel statement. This situation is depicted in Figure 2(b). In this case the two possible orders *(a, b)* and *(b, a)* will be both considered to generate the test skeletons.

- ”same-receiver” pattern. This case relates to a flow in which $receiver(CT_i) = receiver(CT_j) = sender(CT')$ where $i \neq j \wedge i, j \in \{1..n\}$. This situation is depicted in Figure 2(c) where CTs orders *(a, f, g)*, *(f, a, g)*, *(f, g, a)*, *(a, g, f)* and *(g, f, a)* are different orders, all considered relevant for testing derivation purpose. Whereas the order *(g, a, f)* is equivalent to *(g, f, a)*(already considered) and we can omit it.
- ”closed chain” pattern. This case relates to a flow in which some participants are ”sender” and ”receiver” in different CTs so to create a cycle of send-receive relation. The strategy will consider in this case as many orders as the number of tasks. This situation is depicted in Figure 2(d) where sequences *(a, b, e)*, *(b, e, a)* and *(e, a, b)* are considered different traces. Whereas the order *(e, b, a)* is equivalent to *(e, a, b)* (already considered) and we can omit it.

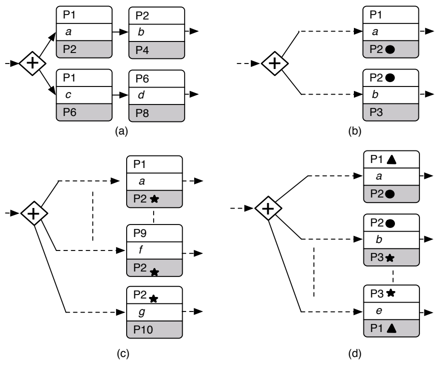


Fig. 2. a) Concurrency of CTs without dependencies; (b) ”receiver-sender”; (c) ”same-receiver”; (d) ”chain” of dependency. Participant are labeled with symbols to show dependencies they belong to.

III. The ParTes strategy

A service that intends to play a role should be tested considering the possible integration scenarios foreseen by the choreography definition. Taking as input a BPMN2 choreography specification, the ParTes (PARticipant TESting) strategy returns test case skeletons suitable to assess if a service can correctly behave when integrated with others.

Our strategy, according to the message dependency defined above, aims to reduce BPMN2 *ParallelGateway* statements to BPMN2 *ExclusiveGateway* statements which include CTs execution traces allowed by the original choreography respecting all the dependencies above defiend. These orders will be re-organized in a tree data structure (*interaction tree*) in which each root-leaf path will represent a possible complete choreography execution trace used to derive test skeletons. This section explains how we reduce nested gateways to simplify the wokflow structure while next section will provide an overview of the reduction of a single *ParallelGateway* that requires to handle message dependency.

Our reduction strategy considers the choreography specification in a formal and technologic-independent way as a *Workflow Graph* [1]. A *workflow graph* $G = (N, T)$ is a simple directed acyclic graph (DAG) where N is a finite set of nodes and T is a finite set of transitions, representing directed edges between two nodes. In this structure for each transition $t \in T$ we define the two mappings $fromNode[t]$ and $toNode[t]$ which return the incoming node and the outgoing node of the transition t respectively. Moreover, for each node $n \in N$ we have:

- $nodeType[n] \in \{TASK, OR, PARALLEL\}$ represents the type of a node. $TASK$ stands for a ChoreographyTask, while OR and $PARALLEL$ stand respectively for *ExclusiveGateways* (decision point) and *ParallelGateways* [2].
- $OutTrans[n] = \{t : t \in T \wedge fromNode[t] = n\}$ and $InTrans[n] = \{t : t \in T \wedge toNode[t] = n\}$ are the sets of outgoing and incoming transitions to/from n
- $OutNodes[n] = \{m : m \in N \wedge \exists t \in T$ s.t. $(fromNode[t] = n \wedge toNode[t] = m)\}$ and $inNodes[n] = \{m : m \in N \wedge \exists t \in T$ s.t. $(toNode[t] = n \wedge fromNode[t] = m)\}$ are the sets of succeeding/preceding nodes that are adjacent to n.
- $din[n]$ and $dout[n]$ are respectively the number of incoming and outgoing transitions to/from n

The workflow graphs we consider in this paper are *well-structured*, i.e. there are matching pairs of nodes that splits and joins the flow. Well-structured workflow graphs are often used for the sake of comprehension [3] and for analysis purpose [4]. Moreover, they do not contain cycles. Their presence in the original choreography will be reduced exploring the cycle for a bounded number of iteration. By default our approach makes a single unfolding of each cycle, nevertheless the tester could decide to unfold each cycle with higher values.

Given a workflow graph, we start a reduction procedure to exert the CT execution traces that are allowed by the relative choreography. To do this we need to isolate fragments of the choreography characterized by a Single Entry and a Single Exit (SESE) boundary nodes that can be easily replaced with other equivalent fragments without parallel processes. This decomposition in fragments can be performed building a Refined Process Structure Tree (RPST) [5] that consists in a hierarchy of sub-workflows. The hierarchy represents an ordering in the fragments to be used in our reduction procedure and we call *reduction set* the set of reducible fragments (that are leaf in the hierarchy) that have as entry and exit point a *ParallelGateway*.

The decomposition and the reduction are taken into account by the REDUCE procedure, reported in the Algorithm 1. REDUCE performs the reduction of *ParallelGateways* contained in the *reduction set* (row 6) and the refactoring of the resulting workflow (rows 7,8) according to a set of rules. After this, a new reduction set is built and the procedure iterates until the wokflow is represented by a single one *ExclusiveGateway*.

The rules used for refactoring are shown in Figure 3. For the sake of comprehension we reported the rules using a BPMN2 notation.

Algorithm 1 REDUCE(G)

```
Input: $G$ is a well-structured workflow graph from BPMN2
Output: a new $G$ with one OR node and a set of sequences
 1: procedure REDUCE($G$)
 2:     Build a reduction set $R$ using an RPST $T$ of $G$
 3:     a fragment $F \in R$ iff $F$ is a leaf of $T$
 4:     while $\forall i \in T, depth(i) = 1 \vee child(child(i)) \neq \emptyset$ do
 5:         for each $F \in R$ do
 6:             REDUCE_PARALLEL_GATEWAY(F,G)
 7:             MERGE(F,G)
 8:             REFACTOR(F, G)
 9:         end for
10:         Build a new reduction set $R$ using a $T$ of $G$
11:     end while
12: end procedure
```

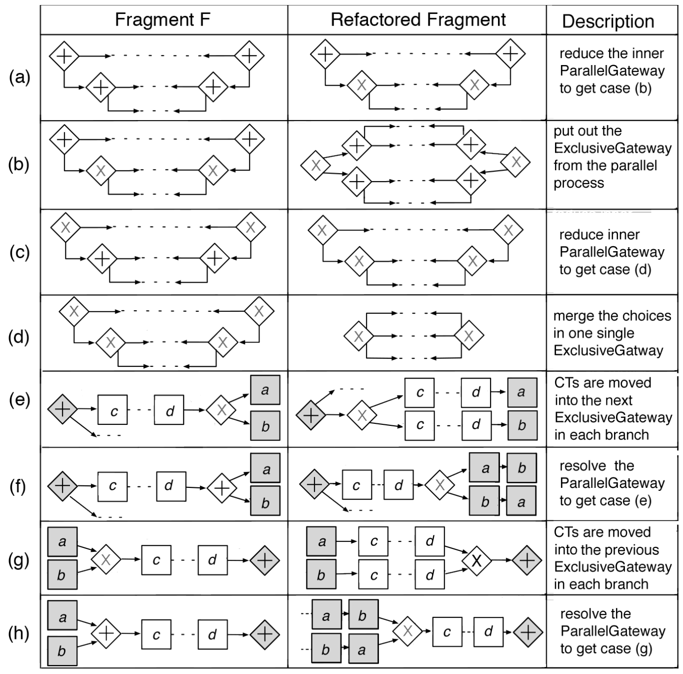

Fragment F
Refactored Fragment
Description
(a)
reduce the inner ParallelGateway to get case (b)
(b)
put out the ExclusiveGateway from the parallel process
(c)
reduce inner ParallelGateway to get case (d)
(d)
merge the choices in one single ExclusiveGatway
(e)
CTs are moved into the next ExclusiveGateway in each branch
(f)
resolve the ParallelGateway to get case (e)
(g)
CTs are moved into the previous ExclusiveGateway in each branch
(h)
resolve the ParallelGateway to get case (g)
a
b
c
d

Fig. 3. Refactoring rules

In particular, subfigures (a) and (c) show a cascade of two gateways where the inner one is a *ParallelGateway*. In this kind of situations the internal gateway will be reduced by REDUCE_PARALLEL_GATEWAY (Algorithm 2) to produce the situations depicted in (b) and (d) respectively, where the inner gateway is an *ExclusiveGateway*. Such situations can be reduced according to the proposed scheme that pushes *ParallelGateways* to internal fragments (rule (b)) to perform their reduction and decrease the number of *ExclusiveGateways* through the application of the rule (d). The real reduction in REDUCE_PARALLEL_GATEWAY is performed by procedures called at rows 2 and 3 of Algorithm 2. These will be analyzed in Section IV. The situations (a) (b) (c) (d) are handled by the REFACTOR procedure (not showed here given the space constraints).

Algorithm 2 REDUCE_PARALLEL_GATEWAY(F)

```
Input: $F$ is a fragment with entry node $u$ and exit node $v$
Output: a new $F$ with PARALLEL replaced by OR
1: procedure REDUCE_PARALLEL_GATEWAY($F$)
2:     $AG \leftarrow CREATE\_GRAPH(pn)$
3:     $TN \leftarrow EXLORE\_GRAPH(AG)$
4:     build a fragment $F'$ from $TN$
5: end procedure
```

Subfigures (e) and (g) show how sequences of *Choreography Tasks* can be moved inside an *Exclusive Gateway* when these are located immediately before the node that opens the fragment or immediately after the node that closes it. Situations depicted in (f) and (h) lead respectively to (e) and (g) after the reduction of parallelism. The situations (e) (f) (g) (h) are handled by the MERGE procedure (not showed here given the space constraints).

IV. Reduction from ParallelGateway to ExclusiveGateway

A. The Affection Graph

The core of the PARTES strategy is the REDUCE_PARALLEL_GATEWAY procedure. The procedure takes into consideration the message dependencies among participants handling the reduction of a single *ParallelGateway* in an efficient way so to consider only those execution traces that can be relevant for testing purpose. So far we considered the result of the reduction in the form of an *ExclusiveGateway* but technically the algorithms returns execution traces organized as a tree structure (*interleavings tree*) made of nodes labeled by the names of explored *CTs*. To obtain this, we need to formalize dependencies as relations and a new structure for them. We define:

Definition 1: A *structural relation* is a relation over *CT*s that appear sequentially in a choreography. Two nodes $a, b \in N$ with $nodeType[a] == nodeType[b] == TASK$ are in *structural relation* $a \Rightarrow b$ if $b \in OutTrans[a]$. This relation must be respected in all the execution traces.

Definition 2: An *affect relation* is a relation over *CT*s that appear in different branches of a parallel statement. Two nodes $a, b \in N$ with $nodeType[a] == nodeType[b] == TASK$ are in *affect relation* $a \rightarrow b$ if they appear in a ”receiver-sender”, ”same-receiver”, or ”chain” pattern as introduced previously. We state that a *affects* b and b *is affected by* a. We call a the *affecter* node, and b the *affected* node.

These relations are used to build a new structure from the workflow graph, called *affections graph*, which contains all the information to manage message dependencies. We will explore it to found all the execution traces to store in the *interleavings tree*. From now, we refer to the following definitions and notations.

Definition 3: An *affections graph* $AG = (CTset, RTset)$ is characterized by:

- $CTset$ which is a finite set of *ChoreographyTasks* that are used as nodes of the graph;

- $RTset = ST \cup AT$ which is a finite set of transitions where ST and AT respectively represent structural and affect relations between two nodes;

Given the AG, for each node $n \in CTset$:

- $StOut[n] = \{m : m \in CTset \wedge \exists t \in ST$ s.t. $(fromNode[t] = n \wedge toNode[t] = m)\}$ are the nodes pointed by outgoing ST transitions from n and $StIn[n] = \{m : m \in CTset \wedge \exists t \in ST$ s.t. $(fromNode[t] = m \wedge toNode[t] = n)\}$ are the nodes pointed by incoming ST transitions to n. Given that we have max($|StOut[n]|$) = max($|StIn[n]|$) = 1, these sets represent the *preceding node* and the *successor node* of n considering only transitions in ST.
- $StIn[n]^* = StIn[n] \cup StIn[m]^*$ where $m \in CTset \wedge \exists t \in ST$ s.t. $(fromNode[t] = m \wedge toNode[t] = n)$ is the set of all the preceding node of n reached by *structural transitions*.
- $AtOut[n] = \{m : m \in CTset \wedge \exists t \in AT$ s.t. $(fromNode[t] = n \wedge toNode[t] = m)\}$ are the nodes pointed by outgoing AT transitions from n and $AtIn[n] = \{m : m \in CTset \wedge \exists t \in AT$ s.t.$(fromNode[t] = m \wedge toNode[t] = n)\}$ are the nodes pointed by incoming AT transitions to n.
- $AtIn[n]^* = AtIn[n] \cup AtIn[m]^*$ where $m \in CTset \wedge \exists t \in AT$ s.t. $(fromNode[t] = m \wedge toNode[t] = n)$ is the set of all the preceding node of n reached by *affect transitions*.

Definition 4: An *interleavings tree* $IT = (TN, TE)$ is a tree where TN is a finite set of nodes each one labeled with one or more *CT*s and TE is a set of edges. For each node $n \in TN$ we will use the notations *child[n]*, *parent[n]*, with the usual meaning over tree structures. Moreover, we use:

- *ct[n]* the list of all the *CT*s that forms the label of n;
- *graph[n]* used in the next algorithms to indicate a copy of AG associated with n.

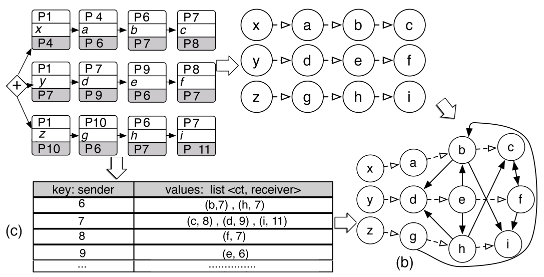
The procedure in Algorithm 3 builds the affection graph AG. The construction considers as nodes of AG each *CT* inside the branches of a *ParallelGateway* (row 2), adds the corresponding structural transition (row 3) and builds an adjacency map of the graph (row 5). We consider this map as an hash table where we can store tuples in the form $\langle sender, [(ct,\ receiver[ct]])\rangle$ where $sender$ is a key. The value stored for a $key = s$ is the list of pairs $(ct,\ receiver[ct])$ such that $sender[ct] = s$. The algorithm proceeds adding the *affect* transitions according to the map. These are added recalling that if two nodes are in the same branch they can not affect each other. In particular, given a pair $(ct,\ receiver[ct])$ (row 7), the ct has affect transitions to all ct'_i such that sender(ct'_i) = receiver(ct) (row 9). This construction is shown in Figure 4.

B. The Affection Graph Exploration

Once we have AG, we need to explore the graph to derive execution traces that replace parallelism. To this aim, the *affection graph* is explored starting each time from a different node. In the exploration each visited node contributes to an execution trace to be added to the *interleavings tree*.

Algorithm 3 Creation of the *affection graph*

```
Input: $pn \in N \mid nodeType[pn] = PARALLEL,\ pn \in F$
Output: $AG$ related to fragment $F$, a dependency map $M$
 1: procedure CREATEGRAPH($pn$)
 2:     $CTset \leftarrow CTset \cup \{n : n \in F\} \backslash \{u, v\}$
 3:     $ST \leftarrow ST \cup \{t : t \in T \wedge \exists n, m \in F | fromNode[t] = n \wedge toNode[t] = m\} \backslash \{OutTrans[u] \cup InTrans[v]\}$
 4:     for $gn \in CTset$ do
 5:         $M \leftarrow M \cup \{\langle sender[gn], (gn,\ receiver[gn])\rangle\}$
 6:     end for
 7:     for a set of $\langle sender, (gn,\ receiver)\rangle \in M$ with the same $sender$ do
 8:         for $x, y \in CTset \mid \exists \langle receiver, (x, y)\rangle$ do
 9:             $AT \leftarrow AT \cup \{t \mid fromNode[t] = gn \wedge toNode[t] = x\}$
10:         end for
11:     end for
12: end procedure
```



key: sender	values: list <ct, receiver>
6	(b,7) , (h, 7)
7	(c, 8) , (d, 9) , (i, 11)
8	(f, 7)
9	(e, 6)
...	....

Fig. 4. (a) The structural transitions built in AG, (b) affect transitions in AG and (c) the adjacency map.

This exploration is performed by the EXPLORE_GRAPH procedure showed by the Algorithm 4. In rows 3 and 4 we have the start of the exploration and the creation of a copy of $graph[tn]$ for each node in the graph belonging to an affect relation. In this way each iteration can use its copy of the graph without side effects on the original structure. Each time a graph node $gn \in CTset$ has been explored following the affect transitions, a corresponding tree node tn is created (row 5) and associated with the copy of $graph[tn]$ (row 6). The tn relative to the first gn of an exploration will be appended to the tree's $root$ (row 7). When a gn has an incoming structural transition, the sequence of all the $StIn[gn]^*$ preceding nodes must be used as label of the relative tn node (row 9). After that, gn itself is added to the execution trace appending its name to the label of tn (row 12). This leads to the label $ct[tn] = StIn[gn]^* \cup gn$. When a node is used to label a tn, it is removed from its private copy of AG. (rows 11, 13). At this point, the exploration proceeds towards $AtOut[gn]$ nodes calling the procedure EXPL_AFFECTED (row 15). Indeed, if gn is not the starting node and has many affecters, it could receive messages from one, none, all or a subset of them. All these cases should be analyzed to cover all the possibilities of interleaving and the exploration should cover all the affecting nodes of gn, i.e. $AtIn[gn]$, in turn $AtIn[AtIn[gn]]$ and so on. A subprocedure in EXPL_AFFECTED provides this exploration of *affecter* nodes to build a piece of a tree to contribute to the *interleavings tree*. Considering the exploration of affecters starting from gn, it should provide a subtree containing all the possible orders of that can be found in $AtIn[gn]^*$. The procedure EXPL_AFFECTED ends when a gn has not outgoing affect transitions and its remaining *CT*s can be just added to the execution trace. In this case the sequence of these nodes is not important, since they don't affect other CTs. Figure 5 shows an affection graph and a full exploration starting from one node.

Algorithm 4 Exploration of the Affection Graph

```
Input: an affections graph AG
Output: an interleavings tree
 1: procedure EXPLORE_GRAPH($AG$)
 2:     build a tree $root$
 3:     for $gn \in$ CTset do
 4:         build and use a copy $C$ of AG
 5:         build a new node $tn \in TN$
 6:         $graph[tn] \leftarrow C$
 7:         $child[root] \leftarrow tn$
 8:         //copy structural transition form $gn$ to $tn$
 9:         $ct[tn] \leftarrow ct[tn] \cup StIn[gn]^*$
10:         //and remove nodes added to the tree
11:         $CTset \leftarrow CTset \setminus StIn[gn]^*$
12:         $ct[tn] \leftarrow ct[tn] \cup \{gn\}$
13:         $CTset \leftarrow CTset \setminus \{gn\}$
14:         if $AtOut[gn] \neq \emptyset$ then
15:             $EXPL\_AFFECTED(tn, gn)$
16:         else
17:             build a new node $nn \in TN$
18:             $child[tn] \leftarrow nn$
19:             for $y \in CTset$ do $ct[nn] \leftarrow ct[nn] \cup \{y\}$
20:             end for
21:         end if
22:     end for
23: end procedure
```

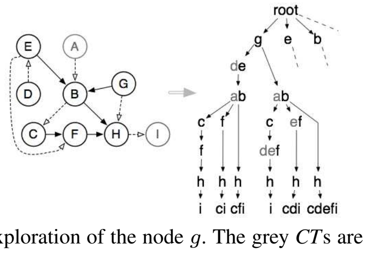


Fig. 5. The full exploration of the node g. The grey *CT*s are those considered looking at structure relation. The tree nodes are depicted with the sequence of *CTs* they represent.

The *interleavings tree* provided by REDUCE_PARALLEL_GATEWAY after the reduction of a single *ParallelGateway*, constitutes a different view of the *ExclusiveGateway* we expected. At this stage the whole choreography is handled by the overall strategy as a tree data structure, re-componing all the *interleavings tree* to build the *interaction tree*, in which each root-leaf path represents a complete possible execution trace which successively leads to test skeletons.

C. An example of reduction

In Figure 6 we show an example of reduction from a BPMN2 diagram containing two nested *ParallelGateways*. The participants that may be affected by others, are marked within the CT for the sake of clarity and the *affection graphs* are showed near the fragments they represents. The RPST decomposition (not shown due to space constraints) produces a *reduction set* composed by the fragments surrounded with dashed lines in the picture.

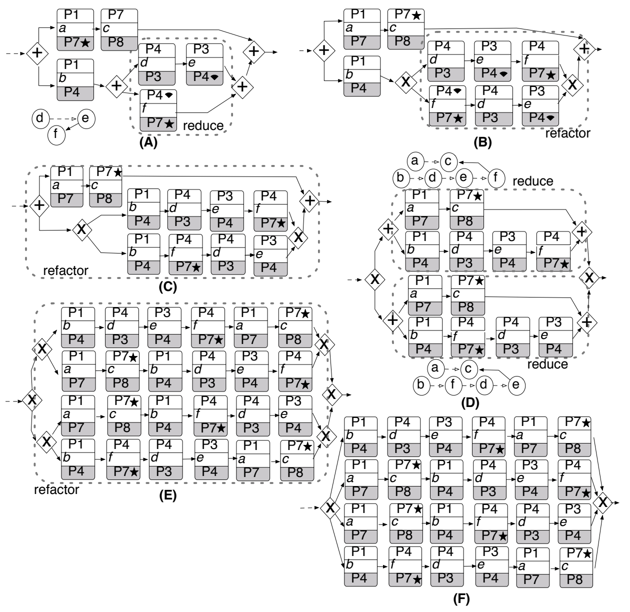


Fig. 6. Example of reduction

Recalling the rules in Figure 3, we show some refactoring steps using a simple example in Figure 6 that involve two nested *ParallelGateway*. Starting from *(A)* the reduction of the inner *ParallelGateway* is applied to obtain the workflow in *(B)* where, for instance, the sequence (d, f, e) is discarded since equivalent to (f, d, e). From *(B)* the rule (e) is applied to merge CT b with the *ExclusiveGateway* and to get to the diagram showed in *(C)*. From *(C)* we apply the rule *(b)* to split the *ParallelGateway* and we get to *(D)*. Two different *ParallelGateways* are now in the diagram and these will be reduced by *REDUCE_PARALLEL_GATEWAY* until *(E)* is obtained. Finally, using the *(c)* rule we obtain *(F)* as the final result. The resulting sequences of *CT*s can now be used to produce test skeletons for the various participants. We have shown this simple example using the BPMN2 notation for clarity. The flow showed in *(A)* can lead to around 5! different interleavings. According to our objectives and considering two message dependencies among the CTs f and e and between *a* and *f*, our strategy returns an *ExclusiveGateway* with a choice of only 4 sequences of CTs covering all the relevant interleavings given the dependencies.

V. From reduction to test skeleton

ParTes generates an *interaction tree* that contains all the execution traces the tester can stress to identify ordering issues about message exchange. Tests will be generated for services willing to play a role within a choreography and we need to isolate the local behavior of a single participant from the tree. We call this operation *projection* in which we isolate local traces of executions, related to a single participant, from the global traces represented by the tree. The local traces are called *trace snippet*. Combining the snippet we build a new tree data structure called *participant interaction tree* where each root-leaf path represents a valid sequence of invocations for a given participant. These traces are the ones we consider relevant for what concerns the detection of interoperability threats and are used to generate test skeletons for each participant. A test skeleton is built reorganizing a *participant interaction tree* to dynamically reconstruct an execution trace which need to be stressed during the test execution. The tester will implement the missing pieces of the generated skeleton adding data information, not available in the choreography diagram or in other BPMN2 diagrams.

We applied the ParTes strategy on a BPMN2 choreography named "Adaptive Customer Relationship Booster", developed within the EU project CHOReOS. It aims to maximize user satisfaction in in-store scenarios inside retail companies, dynamically adapting their marketing strategies according to client's profile. Overall, the specification has 27 CTs, 2 *ExclusiveGateway* and 6 parallel flows. A full exploration would have generated about 10000 execution traces, whereas ParTes provided 84 traces which have been refined into test cases for the ten roles.

VI. Related works

Service oriented computing generally makes difficult both static and dynamic verification. A quite exaustive survey on the topic has been published in [6]. Our work mainly relates to approaches using model based testing strategies. In [7] the authors propose an automatic test case generation for services in a orchestration. The approach exploits the availability of a runnable model and uses model checking to derive test cases suitable to detect possible integration problems.

Graph reduction techniques are an excellent strategy for the analysis of software systems based on processes. In [1] a generic formalization in terms of graphs of a generic language for workflows is given. Our work improve this concepts by applying it to the specific case of choreography diagrams in which each task defines the exchange of messages between two entities. Moreover, we explore tasks looking at message dependencies among them. Workflow graph are used for refactoring and completion of business process models in [8]. The approach use refactoring on graphs to build *well-structured* graphs, and SESE (Single Entry, Single Exit) fragments to verify soundness and to improve the diagnosis information and to fix control-flow errors. Our work exploits the concept of SESE fragments and apply them to choreographies.

A data flow technique to generate tests from service choreography specification has been proposed by [9]. Here a LTS with actions annotated with XPath queries is used to describe the manipulation of messages. A family of data flow testing criteria based on *def-use* associations is proposed. The work focuses on highlighting message manipulation errors and does not intend to relate different message flow as specified by a choreography definition.

VII. Conclusions and Future Works

The paper presented a novel approach to generate test skeleton intended to check the behaviour of services willing to paly a role within a choreography. The derivation strategy takes into account specific dependencies among the communicating tasks so to reduce the possible interleaving to consider. The approach is based on the assumption that no hidden interactions will happen at run-time between the choreography participants. The algorithm is included in a testing tool chain we are developing as part of the EU Project CHOReOS. In such a context we carried on some initial experiments. As future work we intend to extend the experimental part and better elaborate on the characteristics of complexity of the algorithms. At the same time we intend to investigate the possibility of integrating data related information in the exploration and definition of test cases.

Acknowledgements

This work has been supported by the European Project CHOReOS (http://www.choreos.eu) FP7 IP 257178

References

[1] W. Sadiq and M. E. Orlowska, "Analyzing process models using graph reduction techniques," *Information Systems*, vol. 25, no. 2, pp. 117 – 134, 2000, the 11th International Conference on Advanced Information System Engineering.

[2] *Business Process Model And Notation (BPMN) Version 2.0*, Object Management Group, Jan. 2011.

[3] W. M. P. v. d. Aalst, A. Hirnschall, and H. M. W. E. Verbeek, "An alternative way to analyze workflow graphs," in *Proc. of the 14th Int. Conf. on Advanced Information Systems Engineering*, ser. CAiSE '02. Springer, 2002, pp. 535–552.

[4] J. Vanhatalo, H. Völzer, and F. Leymann, "Faster and more focused control-flow analysis for business process models through sese decomposition," in *Proceedings of the 5th international conference on Service-Oriented Computing*, ser. ICSOC '07. Springer, 2007, pp. 43–55.

[5] J. Vanhatalo, H. Völzer, and J. Koehler, "The refined process structure tree," in *Proc. of the 6th International Conference on Business Process Management*, ser. BPM '08. Springer, 2008, pp. 100–115.

[6] M. Bozkurt, M. Harman, and Y. Hassoun, "Testing and verification in service-oriented architecture: a survey," *Software Testing, Verification and Reliability*, 2012.

[7] F. De Angelis, A. Polini, and G. De Angelis, "A counter-example testing approach for orchestrated services," in *Proc. of 3rd ICST 2010*. Washington, DC, USA: IEEE Computer Society, 2010, pp. 373–382.

[8] J. Vanhatalo, H. Vlzer, F. Leymann, and S. Moser, "Automatic workflow graph refactoring and completion," in *Service-Oriented Computing, ICSOC 2008*, ser. LNCS. Springer, 2008, vol. 5364, pp. 100–115.

[9] L. Mei, W. K. Chan, and T. H. Tse, "Data flow testing of service choreography," in *Proc. of the the 7th joint meeting of the European software engineering conference*, ser. ESEC/FSE '09. ACM, 2009, pp. 151–160.