# Message Correlation in Web Services Choreographies: a 4-phase Validation Method

Gregory Van Seghbroeck, Bruno Volckaert, Filip De Turck, Bart Dhoedt
Ghent University - IBBT
Department of Information Technology (INTEC)
Ghent, Belgium
Gregory.VanSeghbroeck@intec.ugent.be

*Abstract*—**The majority of large companies are adopting Service Oriented Architectures, mainly to automate their business processes, both centralized and distributed. This paper will focus on distributed business processes. At the moment there are two interesting ways to implement a distributed business process, via orchestration or choreography. Whereas an orchestration can be thought of as a service composition with a single participant taking the lead, a choreography is a decentralized collaboration between different autonomous participants. One of the most prominent remaining issues, associated with both approaches, is the correlation problem, which is addressed in this paper. We will show that the abstract overall view, provided by a choreography description, makes it possible to determine (even at design time) whether its interactions can be unambiguously correlated. It is shown that this correlation validation is more feasible to realize in case of choreographies than with orchestrations, due to the orchestration's limited view on the overall business process.**

***Correlation, Choreography, WS-CDL***

## I. Introduction

Service Oriented Architectures (SOA) are becoming mainstream in large businesses. The Gartner Group even claims in [1] that the adoption of service-oriented architecture among larger European companies is "nearly universal". These SOAs are in most cases used to streamline the communication inside the company (enterprise application integration, EAI) and to facilitate business-to-business (B2B) communication. Both B2B and EAI applications can be implemented as distributed business processes. Nowadays there are a couple of interesting ways to implement such distributed business processes, the most important are via orchestrations and choreographies. An orchestration uses a central engine to execute a service composition, where the orchestration engine always takes the lead to invoke external services. A choreography is a distributed service collaboration between different autonomous partners.

It is evident that choreographies are best used for designing B2B applications, because every company can be seen as an autonomous partner. However, these B2B applications are very often built by every partner as separate orchestration, hoping they will collaborate perfectly. Orchestrations, with WS-BPEL [2] as its most prominent standard, can rely on a huge set of both commercial and open source tools, runtime environments and even modeling notations such as the Business Process Modeling Notation (BPMN, [3]) and the Unified Modeling Language (UML, [4]). We only can assume that the lack of good tools is slowing down the adoption of choreographies, because choreographies have some huge advantages over orchestrations. The most significant advantage is that a choreography describes the abstract global view of distributed business process, which is, however, also one of its drawbacks. Whereas a WS-BPEL can be immediately deployed and executed, a choreography only describes the interactions between the participants. A lot of effort needs to be made to mould this description into executable code.

This paper presents a validation method that will make it easier for a developer to design choreographies. We will use the very detailed and comprehensive specification, the web services choreography description language (WS-CDL, [5]), to help us in our effort to tackle the message correlation problem. Message correlation is needed to determine for each received message the correct execution instance, a very difficult task given the distributed nature of a choreography. The validation method consists of four phases that can be automated and integrated in a design tool. Firstly, the choreography description in WS-CDL is used to predict how this choreography would be executed, afterwards we determine how the information is transferred between the different participants. In a third phase we investigate which correlations exist within the description, and finally we look whether the description contains enough information to unambiguously correlate all the different choreography interactions.

The remainder of this paper is structured as follows. The next section provides some related work and section 3 illustrates some important concepts of WS-CDL by using a simple example. This example is also used to explain the different phases of the validation method, covered in sections 4, 5 and 6 and 7. Finally, section 8 presents the conclusion.

## II. Related Work

Message correlation is also an issue in orchestrations. Some good solutions to manage message correlation at runtime are used inside orchestration engines, e.g. Apache's ODE [6] uses its Stateful Exchange Protocol based on stateful endpoints and state callbacks. These frameworks and

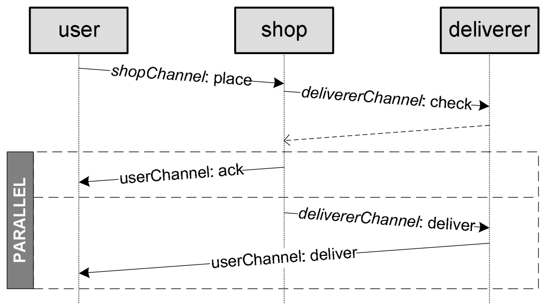

Figure 1. Illustrative example: a simple order management system

engines can only be used at runtime, not the best moment to discover that the message correlation defined in WS-BPEL cannot cope with reality. WS-BPEL allows at least to describe correlation properties and correlation sets, which is not the case for the different modeling languages used to model orchestrations. Neither BPMN, nor UML have a way to enforce message correlation. Design tools such as Intalio's BPMS Designer [7] extend the standard BPMN with WS-BPEL's correlation sets to include a notion of message correlation in these modeling languages.

A lot of research is done on using formal models to validate both orchestrations and choreographies. In [8] and [9] the authors create a mapping between WS-BPEL and respectively π-calculus and Petri-nets to validate an orchestration's workflow. Similar methods have been used for WS-CDL choreographies. In [10] the authors have created a method to check the conformance between on the one hand a choreography description and a participant's implementation using bisimulation, a concept from the π-calculus. In [11] a similar topic is discussed but using Petri-nets.

The message correlation is also addressed in [12]. Here the authors investigate instance isolation in SOAs. They use Petri-net representations of different collaborating orchestrations, which is more a bottom-up approach to the message correlation problem. By using an interconnection model and the correlation sets the loose communication channels are connected to each other. This can of course only work if there exists an unambiguous mapping between the different correlation sets of the collaborating orchestrations. In other cases, manual intervention is necessary. In contrast to this we start from a WS-CDL choreography description in

Given the Variables $xVar$, $yVar$, $\text{s}Var$ and $\text{r}Var$, the RoleTypes $rt$, $from$ and $to$ and $\overline{tokens}$ as the list of Tokens altered or exchanged by the statements:

(i) *Create Copy*: $\mathbf{new}(xVar[\overline{tokens}])_{rt}$

(ii) *Change Copy*: $xVar[\overline{tokens}] \equiv_{rt} yVar$

(iii) *Req. Ex.*: $from \rightarrow to\colon chVar \bowtie op\langle sVar \rightarrow rVar, \overline{tokens}\rangle$

(iv) *Resp. Ex.*: $from \dashleftarrow to\colon chVar \bowtie op\langle sVar \rightarrow rVar, \overline{tokens}\rangle$

Figure 3. The shortened WS-CDL syntax

our correlation validation method, so we already have a well-described interconnection model.

## III. Background on WS-CDL

WS-CDL is an XML-based description language aimed at creating service choreographies by defining, from a global viewpoint, the observable behavior. Besides the observable behavior, accomplished by ordered message exchanges, WS-CDL also describes how the information flows in the choreography. The most important concepts of the WS-CDL specification will be illustrated with a small example: a simple Order Management System (OMS, Fig. 1). We will use this example throughout the entire paper to clarify the different phases in the validation process. Fig. 2 shows the same choreography using the shortened WS-CDL syntax, introduced in [13] (see Fig. 3). For a more elaborate definition of all the WS-CDL concepts we of course refer to the W3C website [5].

The OMS is a collaboration between three partners, which is described in WS-CDL as *RoleTypes*: user, shop and deliverer. However interactions are indeed described between two *RoleTypes*, a channel will be used to transfer the message. A channel, or in WS-CDL terms a *ChannelType*, exposes certain behavior of a *RoleType*. We described three *ChannelTypes* in our OMS choreography, one for each *RoleType*: userChannel, shopChannel and delivererChannel. Both the interactions and the *ChannelTypes* will be responsible for the described global behavior of the choreography.

The information in WS-CDL is held in *Variables*: order and placed are examples in our OMS (Fig. 2). A *Variable* will contain two types of information: application specific data (e.g. the description of the ordered item) and *Tokens*. ON and U are the *Tokens* used in the OMS choreography,

$$
\begin{array}{ll}
(1) & \mathbf{new}(\text{order}[\{\text{ON},\text{U}\}])_{\text{user}} \\
(2) & \text{user} \rightarrow \text{shop}\colon \text{s} \bowtie \mathit{place}\langle \text{order} \rightarrow \text{order}, \{\text{ON},\text{U}\}\rangle \\
(3) & \text{shop} \rightarrow \text{deliverer}\colon \text{d} \bowtie \mathit{check}\langle \text{order} \rightarrow \text{order}, \{\text{ON},\text{U}\}\rangle \\
(4) & \text{resp}[\{\text{ON}\}] \equiv_{\text{deliverer}} \text{order} \\
(5) & \text{shop} \dashleftarrow \text{deliverer}\colon \text{d} \bowtie \mathit{check}\langle \text{resp} \rightarrow \text{resp}, \{\text{ON}\}\rangle \\
\begin{array}{l}(6)\\(7)\\(8)\end{array} & \left(\begin{array}{c|c}
\text{placed}[\{\text{ON}\}] \equiv_{\text{shop}} \text{resp} & \text{delivery}[\{\text{ON},\text{U}\}] \equiv_{\text{shop}} \text{order} \\
\text{placed}[\{\text{U}\}] \equiv_{\text{shop}} \text{order} & \text{shop} \rightarrow \text{deliverer}\colon \text{d} \bowtie \mathit{deliver}\langle \text{delivery} \rightarrow \text{delivery}, \{\text{ON},\text{U}\}\rangle \\
\text{shop} \rightarrow \text{user}\colon \text{d} \bowtie \mathit{ack}\langle \text{placed} \rightarrow \text{placed}, \{\text{ON},\text{U}\}\rangle & \text{deliverer} \rightarrow \text{user}\colon \text{d} \bowtie \mathit{deliver}\langle \text{delivery} \rightarrow \text{delivery}, \{\text{ON},\text{U}\}\rangle
\end{array}\right)
\end{array}
$$

| *RoleType* | user | shop | deliverer |
|---|---|---|---|
| *Channel variable* : *ChannelType* | u : userChannel<br>*.usage* = SHARED<br>*.primary* = {U} | s : shopChannel<br>*.usage* = DISTINCT<br>*.primary* = {ON}<br>*.association* = {U} | d : delivererChannel<br>*.usage* = ONCE<br>*.primary* = {ON}<br>*.association* = {U} |

Figure 2. Shortened WS-CDL choreography description of the order management system

they represent respectively the order number and the user id. *Tokens* are especially used to identify the correct channel instance from the message transferred to a *ChannelType*. This identifying and instantiating process is described using two very important attributes of the *ChannelType*: identities and the usage property. The usage property describes when a new instance of a *ChannelType* will be created. We use the three different ways to define a *ChannelType*'s usage property in the OMS choreography:

- *ONCE*: every time the delivererChannel is targeted in an interaction, a new channel instance will be created.
- *DISTINCT*: a channel instance of the shopChannel may be used multiple times, if the interaction is initiated by the same *RoleType*.
- *SHARED*: the same instance of the userChannel is used by all the *RoleTypes*.

The identities define first of all the set of *Tokens* that will make up this identity and the type of identity. There are four different identity types:

- *Primary*: this identity will be used to uniquely identify a particular channel instance. E.g. a message sent to the userChannel should have a user id $\{\mathrm{U}\}$ *Token* in its content.
- *Alternate*: the alternative identity can also be used to uniquely target a channel instance (there are no examples in the OMS choreography).
- *Association*: via the *association* identity we can correlate different *ChannelTypes*. Let us take the shopChannel for example, whereas the $\{\mathrm{ON}\}$ *Token* will be used to uniquely identify this type of channel, the shopChannel will be associated to the userChannel, because shopChannel's *association* identity is equal to userChannel's *primary* identity.
- *Derived*: this identity will also be used to correlate different *ChannelTypes*. When a channel instance receives a message containing the *Tokens* of a *derived* identity, future channel instances, with a *primary* identity equal to the *derived* identity, will be correlated to the receiving channel instance.

These attributes (a *ChannelType*'s identities and its usage property) will play a significant role in the different phases of the correlation validation method (respectively phase 3 and phase 1). How the pieces of this puzzle fall into place will be outlined in the next sections.

Next to the message exchanges, which describe the choreography's observable behavior, WS-CDL defines three other categories of activities. A first category defines the ordering structures that describe the order in which the embedded activities need to be executed. There are three different ordering structures:

- *Sequence*: every activity need to be executed one after the other.
- *Parallel*: the embedded activities can happen at the same time.
- *Choice*: only one of the embedded activities will eventually be executed. Which activity it will be, is decided at runtime.

Given the *RoleType*s $from$ and $to$, the channel instances $c_i$ and $c_j$, the *Variable*s $sVar$ and $rVar$, and $\overline{tokens}$ as the list of exchanged *Token*s:

(i) *Req.*: $from\colon c_i \rightarrow to\colon c_j \bowtie op\langle sVar \rightarrow rVar, \overline{tokens}\rangle$

(ii) *Resp.*: $from\colon c_i \dashleftarrow to\colon c_j \bowtie op\langle sVar \rightarrow rVar, \overline{tokens}\rangle$

Figure 4. Syntax of the *Extended Exchange* statement

The next category will contain the statements that will create and change *Variables* of a particular *RoleType*, such as the *Create Copy* statement that will be used to initialize *Variables* or parts of *Variables*. WS-CDL also allows a *Change Copy* statement in its choreography description, which will be used to copy parts between two *Variables*. In the scope of this research, we are especially interested in which *Tokens* will be affected by these *Copy* statements. The final category groups the activities that bring no added value to the observable behavior of the choreography. These activities act as placeholders for business functionality or the lack thereof. They will not play any significant role in the correlation validation.

To better comprehend the shortened WS-CDL syntax, we present some examples from the OMS choreography. The first line in Fig. 2 is a *Create Copy* statement where we will initialize two *Tokens*, $\mathrm{ON}$ and $\mathrm{U}$, of the order *Variable*. These changes will only happen on the user *RoleType*. The second line in Fig. 2 shows a *Request Exchange* statement of a user placing (via operation place) an order with a shop. This exchange uses a shopChannel s and will transfer the order *Variable* between the *RoleTypes* user and shop.

## IV. Phase I: Channel Instantiation

As previously indicated WS-CDL only describes the overall choreography and it is not executable in its current form. Unfortunately, to be able to create a correlation validation method, we have to know how the choreography will be executed, because it is during execution that the sent messages need to be correlated to its correct choreography execution. In this phase we automatically predict, given the choreography description, which different instance levels will be created during execution. Reference [13] defines four different instance levels and a method to derive them from a WS-CDL description. The first instance level, the *choreography level*, resembles a particular choreography execution. This virtual level, because even at execution it does not really exist, is a combination of the different *role type instance levels*, the second level. Instance level three, the *channel level*, is also a virtual level, defined for every *ChannelType*. It groups the *channel instance levels*, the fourth instance level, of a particular *ChannelType*. As an extra, [13] also provides an extraction from these *channel instances* to WS-BPEL, which can be implemented, deployed and executed.

The channel instantiation method presented in [13] consists of annotating the different channel instances in the *Exchange* statements of the choreography description. This can of course be done manually, but by implementing the rules presented in [13], it can be incorporated in a design tool. Here the usage attributes of the different *ChannelTypes* will play a prominent role. The method will also add the

$$
\begin{array}{ll}
(1) & \mathbf{new}(\text{order}[\{\text{ON},\text{U}\}])_{\text{user}} \\
(2) & \text{user}{:}\,\mathbf{u_1} \rightarrow \text{shop}{:}\,\mathbf{s_1} \bowtie \mathit{place}\langle \text{order} \rightarrow \text{order}, \{\text{ON},\text{U}\}\rangle \\
(3) & \text{shop}{:}\,\mathbf{s_1} \rightarrow \text{deliverer}{:}\,\mathbf{d_1} \bowtie \mathit{check}\langle \text{order} \rightarrow \text{order}, \{\text{ON},\text{U}\}\rangle \\
(4) & \text{resp}[\{\text{ON}\}] \equiv_{\text{deliverer}} \text{order} \\
(5) & \text{shop}{:}\,\mathbf{s_1} \leftarrow \text{deliverer}{:}\,\mathbf{d_1} \bowtie \mathit{check}\langle \text{resp} \rightarrow \text{resp}, \{\text{ON}\}\rangle \\
\begin{array}{l}(6)\\(7)\\(8)\end{array} & \left(\begin{array}{c|c}
\text{placed}[\{\text{ON}\}] \equiv_{\text{shop}} \text{resp} & \text{delivery}[\{\text{ON},\text{U}\}] \equiv_{\text{shop}} \text{order} \\
\text{placed}[\{\text{U}\}] \equiv_{\text{shop}} \text{order} & \text{shop}{:}\,\mathbf{s_1} \rightarrow \text{deliverer}{:}\,\mathbf{d_2} \bowtie \mathit{deliver}\langle \text{delivery} \rightarrow \text{delivery}, \{\text{ON},\text{U}\}\rangle \\
\text{shop}{:}\,\mathbf{s_1} \rightarrow \text{user}{:}\,\mathbf{u_1} \bowtie \mathit{ack}\langle \text{placed} \rightarrow \text{placed}, \{\text{ON},\text{U}\}\rangle & \text{deliverer}{:}\,\mathbf{d_2} \rightarrow \text{user}{:}\,\mathbf{u_1} \bowtie \mathit{deliver}\langle \text{delivery} \rightarrow \text{delivery}, \{\text{ON},\text{U}\}\rangle
\end{array}\right)
\end{array}
$$

Figure 5. WS-CDL choreography description of the order management system annotated with its correct channel instances

channel instance that initiates the *Exchange* statements, which results in the *Extended Exchange* statements presented in Fig. 4. Determining this initiating channel instance is not always as simple, especially pinpointing the very first channel instance that will be used in the choreography description is not easy. The *ChannelType* of this initializing channel instance is determined using the *Token* context of its *RoleType*, which is created on its part using the preliminary *Copy* statements. It has to be of course possible to derive the initializing channel instance from this context. This gives us a first preliminary validation rule:

**VAL 1** The initializing *RoleType*'s *Token* context needs to contain enough information to distill unambiguously the initializing channel instance.

Fig. 5 shows the choreography description of the order management system after channel instantiation. Notice the annotated channel instances. UserChannel has one instance ($\mathbf{u_1}$), which is evident since its usage attribute is SHARED. There is also one shopChannel instance ($\mathbf{s_1}$), because only the user *RoleType* interacts with this *ChannelType*. Since the usage attribute of the delivererChannel is set to ONCE and there are two interactions that use this *ChannelType*, there will be two different channel instances used in this choreography ($\mathbf{d_1}$ and $\mathbf{d_2}$). The channel instance $\mathbf{u_1}$ is the initializing channel instance and user is the initializing *RoleType*. VAL 1 does not pose any problems, because user's *Token* context contains the *Tokens* $\{\text{ON},\text{U}\}$ and $\{\text{U}\}$ can be used to uniquely identify the userChannel.

## V. Phase II: Information flow

A choreography description provides an abstract overall view on a service collaboration. Not only the interactions are described, but also how the information flows through the choreography. This is a huge advantage over WS-BPEL orchestrations, where we only know how we send and receive information from external partners, but we do not know what happens with this information. We will use this ability to determine the information flow, the second phase of the correlation validation method. This phase can also be fully automated and used while designing the choreography.

### A. Defining the information context

In section 2 we already mentioned that *Variables* are used to hold the information in a choreography description. According to the WS-CDL specification, variables used in a choreography are only known and kept locally by the individual *RoleTypes*, even though it is possible to define the same variable for multiple *RoleTypes*. This means that the information held in these variables can vary and change autonomously for different *RoleTypes*. We introduce for every *RoleType* an information context, which will locally store these variables. By exchanging messages – also information – between *RoleTypes*, it is possible to align these different information contexts. We will use the following symbol to represent the information context of a *RoleType* $rt$: $\mathcal{V}[rt]$, and $\mathcal{V}[rt][xVar]\left\lceil\overline{tokens}\right\rceil$ represents the variable $xVar$ from $rt$'s information context. With this notation, we also know that the variable holds a set of tokens denoted by $\overline{tokens}$.

### B. Information context altering activities

To determine the information flow, the tokens, used in the choreography, are annotated with a counter. What really happens, is that the different token instances get highlighted. Every time a new token instance is created, the counter will be incremented. It is obvious that the activities that have an influence on a *RoleType*'s information context are on the one hand the two *Copy* statements (Create and Change), and on the other hand the *Exchange* statements (Request and Response).

#### 1) The Copy statements

The *Copy* statements will only modify a particular *RoleType*'s information context, indicated by the subscript $rt$ in Fig. 3. The *Create Copy* statement is used to initialize the different tokens that are altered in the statement. Every token from $\mathcal{V}[rt][xVar] \cap \overline{tokens}$ will get a new annotation equal to the incremented counter. The *Change Copy* statement will make sure that $\mathcal{V}[rt][xVar]$'s tokens are annotated with the same value as $\mathcal{V}[rt][yVar]$'s tokens. Here we have to make a couple of remarks. First of all, it is possible that the variables have more tokens than the ones represented in the set $\overline{tokens}$, but only the tokens from this set are annotated. Secondly, it is not allowed to use tokens that are not initialized yet (i.e. tokens that have no annotation value). This last remark gives us a second preliminary validation rule:

**VAL 2** It is not allowed to read from uninitialized tokens.

#### 2) The Extended Exchange statements

To align different *RoleType*'s information contexts, we use the *Extended Exchange* statements, which transfer information between two *RoleTypes*. After completion, we can be sure that the tokens in both information contexts are annotated with the same values. Which annotation values are copied, depends on the direction of the statement: (i) a

TABLE I. THE STATEMENTS IMPACTED BY THE READ/WRITE LOCKING MECHANISM

| Statement | Tokens will get READ lock | Tokens will get WRITE lock |
|---|---|---|
| Create Copy | | $\mathcal{V}[rt][xVar]$ |
| Change Copy | $\mathcal{V}[rt][yVar]$ | $\mathcal{V}[rt][xVar]$ |
| Request Exchange | $\mathcal{V}[from][sVar]$ | $\mathcal{V}[to][rVar]$ |
| Response Exchange | $\mathcal{V}[to][sVar]$ | $\mathcal{V}[from][rVar]$ |

*Request* statement copies the tokens' annotations from $\mathcal{V}[from][sVar]$ to $\mathcal{V}[to][rVar]$, and (ii) a *Response* statement will copy them from $\mathcal{V}[to][sVar]$ to $\mathcal{V}[from][rVar]$. The same remarks and validation rule as with the *Change Copy* statement apply here as well.

## C. READ/WRITE locking mechanism

An interesting choreography contains of course more than one activity. These different activities are combined using the ordering structures defined in the WS-CDL specification. The *Sequence* does not pose any problems: all embedded activities will be handled one by one. The other ordering structures (*Parallel* and *Choice*), however, require special attention.

### 1) The Parallel ordering structure

The *Parallel* ordering structure can be used to describe that the embedded activities need to be executed concurrently. If these concurrent activities alter the same variable from the same *RoleType*'s information context, we cannot deterministically know after the *Parallel* which version will be available in the information context. To cope with this and similar problems, we introduce a READ/WRITE locking mechanism on the tokens within the variables of the different information contexts. The tokens that will receive a READ or a WRITE lock are represented in Table 1 and they need to follow the next principles:

**READ**: a READ lock can be acquired if and only if a token has no WRITE lock set by a previous branch and does not need to be acquired if the active branch already has a lock on this token.

**WRITE**: a token can acquire a WRITE lock if and only if the token only has a lock acquired in the same branch or in an enclosing branch.

Special to the *Parallel* structure is that when a particular branch is completely evaluated, all the acquired locks will stay in place. This means that different parallel branches cannot read tokens that have been altered by other branches and it is not possible to write to tokens of the same variable from the same information context by different parallel branches. We also notice that the READ/WRITE locks need to be aware of its acquiring branch, which is a necessary extension on commonly used READ/WRITE locking mechanisms (e.g. used in databases). Whenever a *Parallel* structure is completely evaluated, the locks acquired within this structure will be removed, because after the structure finishes, we can be sure that all context changes have been carried out.

### 2) The Choice ordering structure

The *Choice* structure, which describes a nondeterministic choice between several activities, also poses problems when the different branches alter the same *RoleType*'s information context. In contrast with the *Parallel* structure, the problem does not manifest itself between the different branches, but after the *Choice* structure. From the WS-CDL description we cannot determine which branch will be taken at execution, so we cannot make assumptions on which information context changes will be carried out. The solution lies also in using the same READ/WRITE locking mechanism as shown in Table 1, but between evaluating the different branches the locks obtained during such an evaluation will be removed. We will, however, save the WRITE locks, because they will be reinstated when all branches have been evaluated and thus the evaluation of the *Choice* structure is finished. In this way we prohibit that variables written in a choice branch will be used further on in the choreography.

This method of discovering the information flow provides us a third preliminary validation rule:

**VAL 3** Any violation against the READ/WRITE locks will cause the validation to fail.

## D. Example

We show the information flow in our illustrative choreography. (1) in Fig. 5 is a *Create Copy* statement, so new tokens will be created for $\mathcal{V}[\text{user}][\text{order}]$. Since the order variable has two tokens used in (1), the annotation counter will be incremented twice, so the information context of *RoleType* $user$ will now contain $\text{order}[\{\text{ON}_1, \text{U}_2\}]$. Line (2) will transfer the two tokens of the order variable from *RoleType* user to *RoleType* shop, which means that now $\mathcal{V}[\text{shop}]$ also contains $\text{order}[\{\text{ON}_1, \text{U}_2\}]$. Line (3) will also exchange the order variable, but now between shop and deliverer, thus $\mathcal{V}[\text{deliverer}]$ will also contain $\text{order}[\{\text{ON}_1, \text{U}_2\}]$. The *Change Copy* statement in line (4) will copy the token from $\mathcal{V}[\text{deliverer}][\text{order}]$ to $\mathcal{V}[\text{deliverer}][\text{resp}]$, . This resp variable is then responded back to the shop *RoleType*. After line (5) $\mathcal{V}[\text{shop}]$ contains besides its previous order variable also the variable $\text{resp}[\{\text{ON}_1\}]$. Now that we enter the *Parallel* ordering structure, we can illustrate how the READ/WRITE locking mechanism works. First of all we start with the left branch. Line (6/L) will copy the token annotation value from the variable resp to the variable placed, which means that they respectively get a READ and a WRITE lock. In line (7/L) a token from the order variable is copied into placed, so first of all, order acquires a READ lock and placed should get a WRITE lock. But placed already has a WRITE lock. Luckily this lock was acquired in the same branch of the *Parallel* ordering structure, so we can carry on with determining the information flow. Line (8/L) will transfer the entire placed variable from the shop to the user *RoleType*, which copies the annotations of $\mathcal{V}[\text{shop}][\text{placed}]$ to $\mathcal{V}[\text{user}][\text{placed}]$. The placed variable will get a WRITE lock in user's information context and a READ lock in the information context of shop, however, since there is already a WRITE lock on this variable in $\mathcal{V}[\text{shop}]$, nothing changes here. With line (8/L) we have completed the left branch and we go on to the next branch. Since we are in a *Parallel* ordering structure, all the locks are

$$\begin{array}{ll}
(1) & \mathbf{new}(\mathrm{order}[\{\mathrm{ON}_1,\mathrm{U}_2\}])_{\mathrm{user}} \\
(2) & \mathrm{user}{:}\,\mathbf{u_1} \rightarrow \mathrm{shop}{:}\,\mathbf{s_1} \bowtie \mathit{place}\langle \mathrm{order} \rightarrow \mathrm{order}, \{\mathrm{ON}_1,\mathrm{U}_2\}\rangle \\
(3) & \mathrm{shop}{:}\,\mathbf{s_1} \rightarrow \mathrm{deliverer}{:}\,\mathbf{d_1} \bowtie \mathit{check}\langle \mathrm{order} \rightarrow \mathrm{order}, \{\mathrm{ON}_1,\mathrm{U}_2\}\rangle \\
(4) & \mathrm{resp}[\{\mathrm{ON}_1\}] \equiv_{\mathrm{deliverer}} \mathrm{order} \\
(5) & \mathrm{shop}{:}\,\mathbf{s_1} \dashleftarrow \mathrm{deliverer}{:}\,\mathbf{d_1} \bowtie \mathit{check}\langle \mathrm{resp} \rightarrow \mathrm{resp}, \{\mathrm{ON}_1\}\rangle \\
\begin{array}{l}(6)\\(7)\\(8)\end{array} & \left(\begin{array}{c|c}
\mathrm{placed}[\{\mathrm{ON}_1\}] \equiv_{\mathrm{shop}} \mathrm{resp} & \mathrm{delivery}[\{\mathrm{ON}_1,\mathrm{U}_2\}] \equiv_{\mathrm{shop}} \mathrm{order} \\
\mathrm{placed}[\{\mathrm{U}_2\}] \equiv_{\mathrm{shop}} \mathrm{order} & \mathrm{shop}{:}\,\mathbf{s_1} \rightarrow \mathrm{deliverer}{:}\,\mathbf{d_2} \bowtie \mathit{deliver}\langle \mathrm{delivery} \rightarrow \mathrm{delivery}, \{\mathrm{ON}_1,\mathrm{U}_2\}\rangle \\
\mathrm{shop}{:}\,\mathbf{s_1} \rightarrow \mathrm{user}{:}\,\mathbf{u_1} \bowtie \mathit{ack}\langle \mathrm{placed} \rightarrow \mathrm{placed}, \{\mathrm{ON}_1,\mathrm{U}_2\}\rangle & \mathrm{deliverer}{:}\,\mathbf{d_2} \rightarrow \mathrm{user}{:}\,\mathbf{u_1} \bowtie \mathit{deliver}\langle \mathrm{delivery} \rightarrow \mathrm{delivery}, \{\mathrm{ON}_1,\mathrm{U}_2\}\rangle
\end{array}\right)
\end{array}$$

Figure 6. The order management system with the annotated tokens representing the information flow

kept into place. The *Change Copy* statement in line (6/R) will copy the tokens' annotations from $\mathcal{V}[\mathrm{shop}][\mathrm{order}]$ to $\mathcal{V}[\mathrm{shop}][\mathrm{delivery}]$, putting a WRITE lock on the delivery variable. We also try to acquire a READ lock for the order variable, which is possible regardless of the already present READ lock. The next lines in the right branch, (7/R) and (8/R), will pass $\mathcal{V}[\mathrm{shop}][\mathrm{placed}][\{\mathrm{ON}_1,\mathrm{U}_2\}]$ from the shop *RoleType* to the deliverer *RoleType* and afterwards to the user *RoleType*. This poses no problems whatsoever to the locking mechanism. For both the deliverer and the user information context, the delivery variable will get its WRITE lock. Fig. 6 shows the entire choreography description with the annotated tokens.

## VI. Phase III: Correlation rules

Now that we have a choreography description with the different channel instances and the entire information flow annotated, we will present phase three of the correlation validation method. First we will define a set of correlation rules that will be used to evaluate the choreography description. These rules are created conform to the WS-CDL specification by strictly interpreting the specification. During the evaluation we create a set of ordered pairs (further referred to as $\mathcal{S}$) indicating a correlation relation. Each ordered pair $(x, y)$ is represented in the correlation rules as: $y$ **corr** $x$, meaning that, given a particular choreography description, $y$ can be correlated to $x$. Both $x$ and $y$ are elements of $\mathcal{C} \cup \mathcal{E} \cup \chi$, with $\mathcal{C}$ being the set of all used channel instances, $\mathcal{E}$ the set of *Extended Exchange* statements and $\chi$ the choreography instance.

### A. Message exchange correlation

Let us start with the easiest correlation rules, the ones where we will correlate an *Exchange* statement to its corresponding channel instance. In section 2 we already mentioned there are two ways to uniquely identify a *ChannelType*: via a *primary* or an *alternate* identity. Messages sent to a particular *ChannelType* will have to include the tokens of one of these identities to reach the correct channel instance, in other words, to make it possible to correlate the message or the exchange to that channel instance. In the following correlation rules we will represent the evaluated *Exchange* statement by $E$ and the annotated tokens representing the *primary* or the *alternate* identities as respectively the set $PRIM$ or the set $ALT$. The other identities defined in WS-CDL will get a similar notation: $ASSOC$ and $DER$ for respectively *association* identity and *derived* identity. These identities are properties of the channel instance used in the correlation rule. The used channel instance depends on whether $E$ is a *Request* or a *Response* statement. A request message has to include the identity tokens of the target channel instance, which is obvious. With the response message it is not as trivial. The WS-CDL specification dictates that a response message needs to contain the identity information of the *ChannelType* the response is exchanged on, which is equivalent to the channel instance that sends the response. This actually means

$$\frac{\mathcal{S} = \emptyset \wedge PRIM \subseteq from.context}{\mathcal{S} = \{c_i^{PRIM} \, \mathbf{corr} \, \chi^{\mathcal{T}}\}} \qquad \text{[INST CHOR]}$$

$$\frac{PRIM \subseteq \overline{tokens}}{\mathcal{S} = \mathcal{S} \cup \left\{E \, \mathbf{corr} \, c_j^{PRIM}\right\}} \qquad \text{[PRIM CORR]}$$

$$\frac{PRIM \nsubseteq \overline{tokens} \wedge ALT \subseteq \overline{tokens}}{\mathcal{S} = \mathcal{S} \cup \left\{E \, \mathbf{corr} \, c_j^{ALT}\right\}} \qquad \text{[ALT CORR]}$$

$$\frac{\left(\nexists \, y \, \mathbf{corr} \, c_j^{ALT} \in \mathcal{S}\right) \wedge \left(PRIM \cup ALT \subseteq \overline{tokens}\right)}{\mathcal{S} = \mathcal{S} \cup \left\{c_j^{ALT} \, \mathbf{corr} \, c_j^{PRIM}\right\}} \qquad \text{[ALT MATCH]}$$

$$\frac{\left(\exists \, y \, \mathbf{corr} \, x \in \mathcal{S},\ y = z^{ASSOC} \vee x = z^{ASSOC}\right) \wedge \left(\exists_1 \, y \, \mathbf{corr} \, c_j^{PRIM} \in \mathcal{S}\right) \wedge \left(PRIM \cup ASSOC \subseteq \overline{tokens}\right)}{\forall \, y \, \mathbf{corr} \, x^{ASSOC} \in \mathcal{S},\ \mathcal{S} = \mathcal{S} \cup \left\{c_j^{PRIM} \, \mathbf{corr} \, x^{ASSOC}\right\}} \qquad \text{[ASSOC MATCH]}$$

$$\frac{PRIM \cup DER \subseteq \overline{tokens}}{\forall \, c \in \mathcal{C}[primary \equiv DER],\ \left(\nexists \, y \, \mathbf{corr} \, c^{DER} \in \mathcal{S}\right) \rightarrow \left(\mathcal{S} = \mathcal{S} \cup \left\{c^{DER} \, \mathbf{corr} \, c_j^{PRIM}\right\}\right)} \qquad \text{[DER MATCH]}$$

Figure 7. Correlation rules: need to be evaluated in the presented order

TABLE II. CORRELATION RULES APPLIED TO THE ORDER MANAGEMENT SYSTEM

| Exchange statement | Applied rule | Set $\mathcal{S}$ |
|---|---|---|
| $E_2$ | INST CHOR<br>PRIM CORR<br>ASSOC MATCH | $\mathrm{u}_1^{\{\mathrm{U}_2\}}$ **corr** $\chi^{\{\mathrm{ON}_1,\mathrm{U}_2\}}$<br>$E_2$ **corr** $\mathrm{s}_1^{\{\mathrm{ON}_1\}}$<br>$\mathrm{s}_1^{\{\mathrm{ON}_1\}}$ **corr** $\mathrm{u}_1^{\{\mathrm{U}_2\}}$ |
| $E_3$ | PRIM CORR<br>ASSOC MATCH | $E_3$ **corr** $\mathrm{d}_1^{\{\mathrm{ON}_1\}}$<br>$\mathrm{d}_1^{\{\mathrm{ON}_1\}}$ **corr** $\mathrm{u}_1^{\{\mathrm{U}_2\}}$ |
| $E_5$ | PRIM CORR | $E_5$ **corr** $\mathrm{d}_1^{\{\mathrm{ON}_1\}}$ |
| $E_{8L}$ | PRIM CORR | $E_{8L}$ **corr** $\mathrm{u}_1^{\{\mathrm{U}_2\}}$ |
| $E_{7R}$ | PRIM CORR<br>ASSOC MATCH | $E_{7R}$ **corr** $\mathrm{d}_2^{\{\mathrm{ON}_1\}}$<br>$\mathrm{d}_2^{\{\mathrm{ON}_1\}}$ **corr** $\mathrm{u}_1^{\{\mathrm{U}_2\}}$ |
| $E_{8R}$ | PRIM CORR | $E_{8R}$ **corr** $\mathrm{u}_1^{\{\mathrm{U}_2\}}$ |

that we always will need to correlate statement $E$ with channel instance $c_j$. Besides creating the correlated pairs, we will give the channel instances an extra annotation, the set of tokens representing the used identity ($PRIM$, $ALT$, $ASSOC$ or $DER$), which will help us later with the validation. The formal versions of these two correlation rules, [PRIM CORR] and [ALT CORR] are shown in Fig. 7.

## B. *Channel instance correlation rules*

Using the previous two correlation rules we can correlate a particular *Exchange* statement with its corresponding channel instance. Notice that exchanges correlated via their *primary* or via their *alternate* identities will result in different annotations on the channel instance, even for equal channel instances. According to WS-CDL an alternate identity can only be used to identify a particular channel instance if in a previous exchange this alternate identity appears together with the correct primary identity. This rule is formally defined in Fig. 7 as [ALT MATCH].

The other identity types defined by WS-CDL, *association* and *derived* identities, are used to correlate different channel instances to each. The *association* identity is used to correlate the channel instance, which defines the identity, with previous evaluated channel instances. The latter channel instances need to have a *primary* or an *alternate* identity defined equal to the *association* identity. WS-CDL poses another limitation on the use of an *association* identity: it only has an effect when it is used the first time a channel instance is evaluated. The *derived* identity is used as a future reference, i.e. the channel instances with a *derived* identity can be used to correlate future channel instances. Fig. 7 also has a formal representation for the *association* and *derived* identities ([ASSOC MATCH] and [DER MATCH]). In [DER MATCH] we have a new notation that needs some clarification: $c \in \mathcal{C}[primary \equiv DER]$, which means that $c$ is a channel instance, used within this choreography description, and its primary identity matches the set of tokens $DER$.

## C. *Choreography instance correlation rule*

Now that we can correlate *Exchange* statements to channel instances and we can tie different channel instances to each other, only the link with the choreography instance level remains. This link is provided by the very first channel instance that is used in the choreography. In the channel instantiation phase (section 3, [13]) we determined this channel instance by creating a token context for the first *RoleType* instance of the choreography (please do not confuse it with the information context). We will annotate the channel instance using its *primary* identity tokens. The choreography instance (symbolized by $\chi$) will be annotated with all the different token instances used in this choreography (we will refer to this set as $\mathcal{T}$). $\mathcal{T}$ represent the token instances that need to be uniquely defined for this choreography description. The formal representation of this rule is shown in Fig. 7: rule [INST CHOR].

## D. *Iterated evaluation*

In the previous subsections the correlation rules are defined. These rules only apply to the *Extended Exchange* statements of a choreography description. A particular statement need to be matched against all the rules presented in Fig. 7, in the given order. This matching process needs to be iterated until no changes to $\mathcal{S}$ occur, because some rules may result in correlated pairs that can be used by other rules.

## E. *Example*

As an example, we will build the set $\mathcal{S}$ for our order management system choreography. We start from Fig. 6 with the annotated channel instances and the annotated information flow. As mentioned, the correlation rules only use the *Exchange* statements from the choreography description. We will refer to each statement by its line number, thus the first *Exchange* statement we have to evaluate, is the one on line (2): $E_2 \equiv \mathrm{user}{:}\,\mathbf{u_1} \to \mathrm{shop}{:}\,\mathbf{s_1} \bowtie$ $place\langle \mathrm{order} \to \mathrm{order}, \{\mathrm{ON}_1, \mathrm{U}_2\}\rangle$. The first rule that is matched by $E_2$ is [INST CHOR]: the set $\mathcal{S}$ still is empty and $\mathrm{user}.context = \{\mathrm{ON}_1, \mathrm{U}_2\}$ (we know this from line (1) and the channel instantiation phase), $c_i = \mathbf{u_1}$ and $PRIM = \{\mathrm{U}_2\}$ ($\mathbf{u_1}$ is a channel instance of type *userChannel* and its primary identity only contains the U token), so we will initiate $\mathcal{S}$ as $\left\{\mathrm{u}_1^{\{\mathrm{U}_2\}} \textbf{corr}\ \chi^{\{\mathrm{ON}_1,\mathrm{U}_2\}}\right\}$. $E_2$ can also be matched by [PRIM CORR]: $c_j = \mathbf{s_1}$ and $PRIM = \{\mathrm{ON}_1\}$ ($\mathbf{s_1}$ is of type *shopChannel* and has a primary identity containing the ON token), so this rule will add $\left\{E_2\ \textbf{corr}\ \mathrm{s}_1^{\{\mathrm{ON}_1\}}\right\}$ to the set $\mathcal{S}$. The last rule that can be matched is [ASSOC MATCH]: $ASSOC = \{\mathrm{U}_2\}$ ($\mathbf{s_1}$, a *shopChannel*, has an association identity containing the U token), there exists a channel instance annotated with $\{\mathrm{U}_2\}$ (the pair created by the [INST CHOR] rule) and this is the same statement that initialized $\mathbf{s_1}$, so $\left\{\mathrm{s}_1^{\{\mathrm{ON}_1\}}\ \textbf{corr}\ \mathrm{u}_1^{\{\mathrm{U}_2\}}\right\}$ will be added to $\mathcal{S}$. Other iterations are not necessary anymore, because they will not make any changes to $\mathcal{S}$. Table 2 shows for each *Exchange* statement the rules that are applied and the ordered pairs that will be added to $\mathcal{S}$.

## VII. Phase IV: Correlation Validation

### A. Transitive closure

Once the set $\mathcal{S}$ is completely created, the real validation process can start. The set $\mathcal{S}$ represents, given the choreography description, the entities that can be correlated to each other. The final validation phase checks whether every *Exchange* statement can be correlated to $\chi^{PRIM}$, the choreography instance level. To achieve this, we use the transitive closure of the binary relation depicted by $\mathcal{S}$. The transitive closure of a binary relation $\mathcal{R}$ on a set $\mathcal{X}$ is the minimal transitive relation $\mathcal{R}'$ on $\mathcal{X}$ that contains $\mathcal{R}$. Thus $a\,\mathcal{R}'b$ for any elements $a$ and $b \in \mathcal{X}$ provided there exist $c_0, \dots, c_n$ with $c_0 = a, c_n = b$ and $\forall 0 \leq r < n: c_r\,\mathcal{R}\,c_{r+1}$. The transitive closure on $\mathcal{S}$, we will refer to it as $\mathcal{S}^+$, relates all the entities that can be correlated to each other in a transitive fashion. If we now look for the set of entities that are related to $\chi^{\mathcal{T}}$, then for the choreography to be unambiguously correlatable, this set needs to contain all *Exchange* statements described in the choreography.

**VAL 4** Given $\boldsymbol{\mathcal{E}}$ the set of *Exchange* statements, $\boldsymbol{\mathcal{S}^+}$ the transitive closure of $\boldsymbol{\mathcal{S}}$ and $\boldsymbol{\chi^{\mathcal{T}}}$ the choreography instance level: $\boldsymbol{\mathcal{E} \subseteq \mathcal{S}^+(\chi^{\mathcal{T}})}$

### B. Example

If we want to validate our example choreography, we need to evaluate $\mathcal{S}^+(\chi^{\{\mathrm{ON}_1,\mathrm{U}_2\}})$. This can be easily solved with for example the Floyd–Warshall algorithm [14], a simple and very well-known algorithm in graph theory – the transitive closure can also be used to discover reachability in directed graphs. $\mathcal{S}^+(\chi^{\{\mathrm{ON}_1,\mathrm{U}_2\}}) = \{\mathrm{u}_1^{\{\mathrm{U}_2\}}, \mathrm{s}_1^{\{\mathrm{ON}_1\}}, \mathrm{d}_1^{\{\mathrm{ON}_1\}}$, *E8L,* d*2*ON1*, E8R, E2, E3, E5, E7R*. And as we can see, every *Exchange* statement is part of this set, which means that every statement somehow can be correlated to the same choreography instance level ($\chi^{\{\mathrm{ON}_1,\mathrm{U}_2\}}$).

## VIII. Conclusion

This paper shows how the WS-CDL specification, a comprehensive description language for service choreographies, can be used to validate whether all the interactions can be unambiguously correlated to the same choreography instance. The validation method contains four phases:

- *Channel instantiation phase*: predict how the choreography will be executed at runtime and annotate accordingly the different channel instances.
- *Information flow phase*: determine how the different tokens will transferred between the choreography participants.
- *Correlation rule evaluation phase*: the annotated *Exchange* statements are used to create a set of correlated pairs.
- *Correlation validation phase*: Using the relation, represented by the set from the previous phase, we can determine whether all message exchanges can be correlated to the choreography instance.

We implemented this validation method inside an eclipse based choreography design tool. The presented method allows to determine, whether it is possible for every exchanged message to be correlated correctly at runtime to its meant choreography instance and this while designing the choreography. Something that is not possible even in the more advanced orchestration tools.

## Acknowledgment

Gregory Van Seghbroeck acknowledges the IWT (institute for the promotion of Innovation by Science and Technology in Flanders) for a PhD research grant.

## References

[1] D. Sholler, “2008 SOA User Survey: Adoption Trends and Characteristics”, Gartner Group, Stamford, CT, September, 2008.

[2] D. Jordan, J. Evdemon, et all, “Web Services Business Process Execution Language Version 2.0 (WS-BPEL)”, OASIS, http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html, 2007.

[3] S. A White, et all, “Business Process Modeling Notation Version 1.2 (BPMN)”, Object Management Group, http://www.bpmn.org/, 2009.

[4] D. Baisley, M. Björkander, C. Bock, et all, “Unified Modeling Language Version (UML)”, Object Management Group, http://www.uml.org/, 2009.

[5] N. Kavantzas, D. Burdett, et all, “Web Services Choreography Description Language Version 1.0 (WS-CDL)”, W3C, http://www.w3.org/TR/ws-cdl-10/, November 2005.

[6] M. Riou, et all, “Apache ODE (Orchestration Director Engine)”, Apache Software Foundation, http://ode.apache.org/.

[7] Intalio, “Intalio BPMS Designer”, http://www.intalio.com/.

[8] R. Lucchi and M. Mazzara, “A pi-calculus based semantics for ws-bpel”, Journal of Logic and Algebraic Programming, vol. 70, no. 1, pp. 96-118, January 2007.

[9] C. Ouyang, E. Verbeek, W. M. van der Aalst, S. Breutel, M. Dumas, and Ter, “Formal semantics and analysis of control flow in ws-bpel”, Science of Computer Programming, vol. 67, no. 2-3, pp. 162-198, July 2007.

[10] G. Van Seghbroeck, F. De Turck, B. Dhoedt, and P. Demeester, “Web service choreography conformance verification in M2M systems through the piX-model”, 2007 IEEE International Conference on Pervasive Services, pp. 385-390, July 2007.

[11] W. M. P. van der Aalst, M. Dumas, C. Ouyang, A. Rozinat and H. M. W. Verbeek, “Choreography Conformance Checking: An Approach based on BPEL and Petri Nets”, The Role of Business Processes in Service Oriented Architectures, Dagstuhl Seminar Proceedings, 2006.

[12] G. Decker and M. Weske, “Instance Isolation Analysis for Service-Oriented Architectures”, in IEEE SCC , IEEE Computer Society, pp. 249-256, September 2008.

[13] G. Van Seghbroeck, B. Volckaert, F. De Turck, and B. Dhoedt, “Automated instantiation and extraction of web service choreographies”, International Conference on Internet and Web Applications and Services (ICIW), Italy, Venice, pp. 455-461, May 2009

[14] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, “Introduction to Algorithms, Second Edition”, The MIT Press and McGraw-Hill Book Company, 2001.