Towards a Framework for Detecting Containment Violations in Service Choreography

Faiz UL Muram, Muhammad Atif Javed, Huy Tran and Uwe Zdun
University of Vienna, Faculty of Computer Science,
Software Architecture Research Group, Vienna, Austria
Email: faiz.ulmuram|muhammad.atif.javed|huy.tran|uwe.zdun@univie.ac.at

***Abstract*—In the design and development of service oriented applications, service choreography models describe the interactions between services at different abstraction levels. These models are usually created and evolved independently by different stakeholders and consequently deviations occur among models such as message not received and incompatible behaviours. It is therefore crucial to detect and resolve the deviations before actual implementation and deployment is undertaken. This paper presents a containment checking approach that verifies whether the behaviour (or interactions) described by the local choreography models collectively encompasses those specified in the global model. Previous studies have not considered the containment relationship between global and local choreography models. The proposed approach performs automated transformation of service choreography models into formal descriptions and consistency constraints for leveraging the analytical powers of model checking techniques for the containment verification. The approach provides more informative and comprehensive feedbacks to the stakeholders for identification of containment problems and their resolutions. The applicability of the approach is demonstrated through use case scenarios of ATM machine, travel booking and order processing systems.**

***Keywords*-containment checking; choreography; web services; business process modelling; formal methods.**

I. Introduction

In choreography-based service-oriented systems, a typical design and development scenario is that the global model (aka interaction model) is created by business analysts to agree on interaction scenarios from a global perspective. The global model will then be refined during detailed design phase into the public visible behaviour and hence forms a local choreography model (aka interconnection model) of each participant. The local choreography model shows an abstraction of orchestration internal actions/activities. The local choreography models often deviate from the global model due to the involvement of different stakeholders and independent evolutions. Hence, detecting model inconsistencies in early phases of the service development life cycle is crucial to eliminate as many anomalies as possible before service-oriented systems are actually implemented and deployed.

The literature discusses two possible ways to alleviate such problems: (i) the local models (i.e., representing implementation of individual services) can be generated from the global model [1]; (ii) the global and orchestration models can be created separately and then checked against each other [2]. The former strategy, although helpful to certain extent, did not prevent the overriding of manual changes that are made to complete the models. The later strategy focuses on the assessment of model inconsistencies that require formal descriptions and consistency constraints of the models. However, it is a challenging task to accurately and correctly express such formal descriptions and consistency constraints due to the substantial amount of knowledge and specialized training required for the underlying formalisms and formal techniques. In addition, the produced results are rather cryptic and verbose; they are difficult to interpret and understand for software architects/developers who often have limited knowledge of the underlying formal techniques [3].

In this paper, we proposed a containment checking approach that verifies whether the behaviour (or interactions) described in the local choreography models collectively encompasses those specified in the global model. This improves the quality and correctness of the service oriented systems. To date, however, previous studies have not considered the containment relationship between global and local choreography models. Specifically, we have performed automated translation of global choreography model into consistency constraints i.e., linear temporal logic (LTL) [4] and local choreography models into formal descriptions i.e., SMV language (symbolic model verifier); whereas the NuSMV (new symbolic model verifier) model checker [5] is used that supports the verification of large systems up to 10^{20} states. This way, our approach helps to alleviate the burden of manually encoding consistency constraints, and therefore, increase productivity and avoid potential translation errors. In order to facilitate better feedback, we integrate the counterexample analysis method for locating the cause(s) of containment violations and presenting the appropriate suggestions to stakeholders for their resolutions. Moreover, we investigated the performance of the proposed approach on use case scenarios of ATM machine, travel booking and order processing systems. This is done to ensure whether the stakeholders are supported to verify the containment relationship during their development tasks.

The rest of this paper is organized as follows: Section II motivates the necessity of containment checking in service choreographies and explains a running example modelled using business process model and notation (BPMN) diagrams. Section III describes a novel approach for assessment of containment violations in service choreographies and recommendations for their resolutions. Section IV describes the performance evaluation of the proposed approach. Section V presents the related work. Section VI concludes the paper.

II. Motivation and Running Example

Service choreography is a set of interrelated service interactions at the high-level of abstraction, which represents message exchanges, interaction rules and agreements between web service partners. Figure 1 shows a global model of the *Travel Booking* application modelled using the BPMN 2.0 choreography notation [6]. The sender and receiver of a message are collapsed into one choreography activity; the unshaded and gray shaded bands represent the sender (initiating participant) and the receiver (non-initiating participant) of a message, respectively. The collaboration in travel booking choreography process involves six partners/participants, namely traveller, travel agency, acquirer, airline, hotel and rent a car agencies.

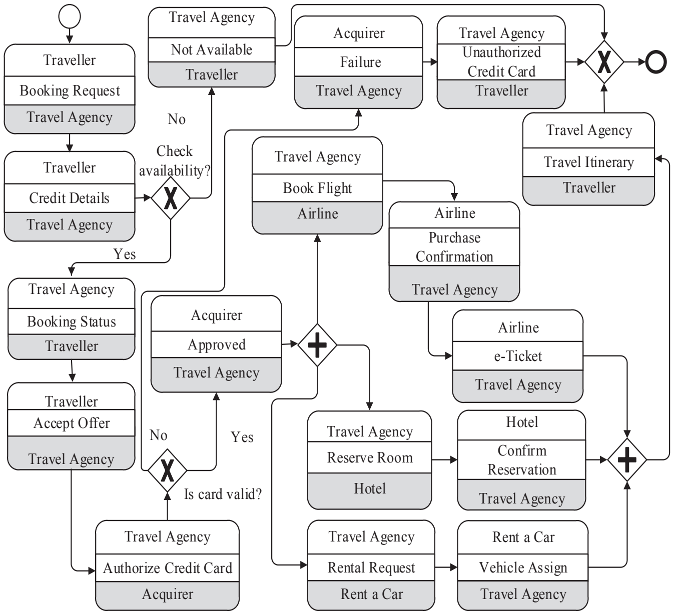


Fig. 1: Travel Booking System: Global Choreography Model

The local choreography models of all participants (i.e., pools) involved in the travel booking system are shown in Figure 2. The message flow indicates the exchange of a message between two participants; whereas the sequence flow reflects the order in which activities are performed within a pool. It is crucial to sequence the choreography activities in such a way that the participants involved in the service choreography know when they are responsible for initiating the interactions. For instance, the `BookingRequest` and `CreditDetails` messages in the global model meant to be received in a sequential order (i.e., CreditDetails message follows BookingRequest message), as shown in Figure 1. However, the local model of the *travel agency* participant shown in Figure 2 specifies that the `CreditDetails` message precedes `BookingRequest` message. Furthermore, the sequential order of `PurchaseConfirmation` and `e-Ticket` messages is replaced by the parallel order using fork and join in the local choreography models, in particulary *travel agency* and *airline*. Please note that the containment checking not only deals with the missing participant or interaction but also misplacement of elements among the models. The undesired containment violations would cause severe problems; for example, improper identification of services and their corresponding service providers, and therefore affect the delivery of services. In order to eliminate such problems, containment checking shall be performed.

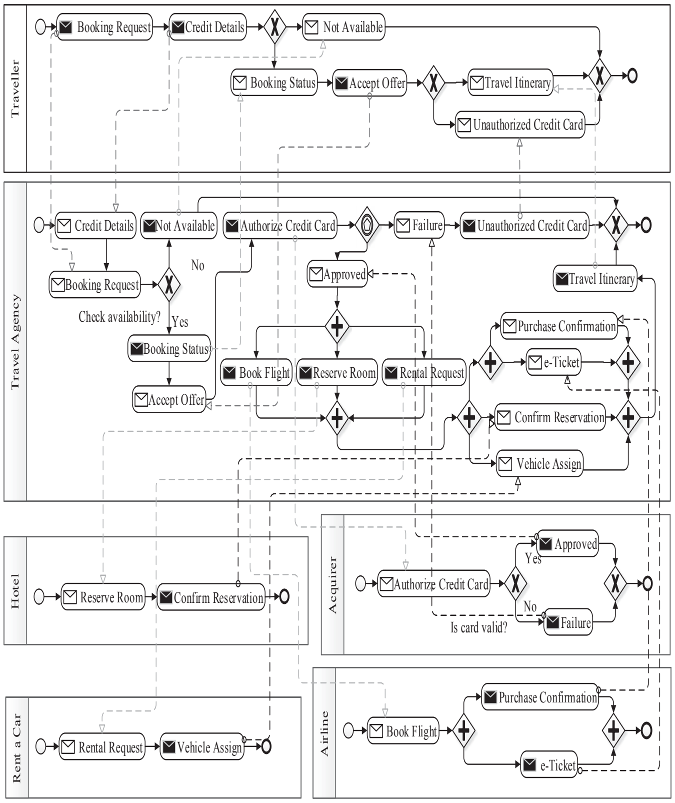
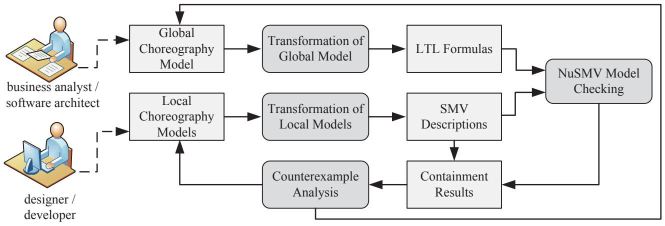


Fig. 2: Travel Booking System: Local Choreography Models

III. Approach

In this section, we address the problem of checking whether the message exchange behaviour (or interactions) described in the joint local choreography models encompasses those specified in the global model. Formally, the containment relationship between service choreographies is defined in such a way that $(GCM \mapsto LTL) \prec (LCMi = (LCM_1 ... LCM_n) \mapsto SMV)$, where *GCM* denotes the global choreography model that is mapped to LTL formulas and *LCMi* denotes a joint set of local choreography models that is mapped to SMV descriptions. An overview of our approach is shown in Figure 3. In the following sections, we describe each step of our approach.



business analyst / software architect
designer / developer
Global Choreography Model
Local Choreography Models
Transformation of Global Model
Transformation of Local Models
LTL Formulas
SMV Descriptions
NuSMV Model Checking
Containment Results
Counterexample Analysis

Fig. 3: Overview of the Containment Checking Approach

A. Generating LTL Constraints from GCM

The section is concerned with the automated transformation of global choreography model (GCM) into formal consistency constraints. From the containment checking perspective, the control flow relations between choreographic activities or interactions need to be represented in an appropriate formalism so that the execution order of interactions will become the consistency constraints for all local choreography models. In this context, a certain execution path is derived from the global model for describing the temporal relationships among the elements (e.g., choreography tasks, senders and receivers of the interactions, and guard conditions) using LTL [4]. The behavioural properties of the model can be easily expressed in LTL. It is widely-used in formal verification tools [7].

LTL extends the classical propositional logic ($\neg$, $\wedge$, $\vee$, $\rightarrow$) with several future temporal operators such as **F** ("**F**inally"), **X** ("ne**X**t"), **G** ("**G**lobally") and **U** ("**U**ntil"), and past temporal operators such as **H** ("**H**istorically"), **O** ("**O**nce") and **Y** ("**Y**esterday"). This research focuses on both future and past temporal operators. The exclusive decision and merge gateways are implemented as "$(a \wedge \neg b) \vee (\neg a \wedge b)$" instead of using logical "`xor`" operator. This is because, xor operator yields `true` not only when one of its operands is `true` but also when the odd numbers (i.e., $n \geq 3$) of the operands are `true` [8]. The generated formulas for different path constructs i.e., fork and join are enclosed by the **G** and **H** operators to express that all possible execution scenarios of the formulas are satisfied.

Algorithm 1 Translate *GCM* into LTL Formulas

```
procedure TRANSLATE($GCM$)
    $Q \leftarrow \varnothing$        ▷ $Q$ is the queue of non-visited interactions
    $V \leftarrow \varnothing$        ▷ $V$ is the queue of visited interactions
    $Q \leftarrow Q \cup$ get_start_events($e$)
    for all $i \in Q$ do
        $V \leftarrow V \cup \{i\}$
        $Q \leftarrow Q \setminus \{i\}$
        generate_ltl_code($i$)
        $I_{succeding\_interactions} \leftarrow$ get_interaction($i$)
        for all $j \in I_{succeding\_interactions}$ do
            if ($j \notin V$) then
                $Q \leftarrow Q \cup \{j\}$
    extracts interaction information;
    binds input values and generates ltl formulas using the following templates:
    for all $(i \geq 0) \wedge V \leftarrow V \cup \{i\}$ do
        if $i \in AND-Join \wedge i \in I_{preceding\_interactions\_rec}$ then
            '''
              (LTLSPEC G(«i» & «i» -> F «j») & H(«j» -> O «i» & «i»))
            '''
```

The construction of LTL formulas for containment checking is a highly knowledge intensive endeavour. In this context, the LTL-based transformation rules are defined to formally represent the constructs of GCM. Therefore, the input GCM is automatically translated into corresponding LTL formulas using the LTL-based transformation rules. In our formalisation, we map a choreography interaction as a 3-tuple $\langle participant\_name, msg, snd/rec \rangle$; where (i) *participant_name* indicates the corresponding participant; (ii) *msg* represents a message that describes communication contents between two participants; and (iii) *snd* and *rec* describe the sending and receiving actions of the corresponding message, respectively. However, the initiating participants of the choreography activities must have been involved in the previous activity (excluding first activity).

The Eclipse Xtend[1] framework is leveraged to translate the GCM into LTL formulas. Specifically, the breadth-first search algorithm is extended with three helper functions, namely `get_events(`e`)`, `get_interaction(`i`)` and `generate_ltl_code(`i`)`, as shown in Algorithm 1. The function `get_events(`e`)` returns a set of start events. A start event indicates the starting point of a choreography. Hence, it has no incoming sequence flow. The function `get_interaction(`i`)` extract all interactions i. The choreography tasks along with the senders and receivers of the messages, as well as the message exchange dependencies (i.e., sequence flows or gateways) are extracted. An interaction j is called "succeeding interaction" of i if there is a control flow going from i to j. Thus, a set of succeeding choreography activities of i can be achieved by following all of its outgoing control flows.

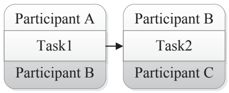
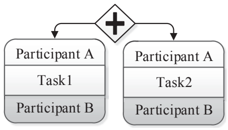
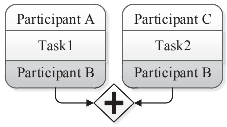
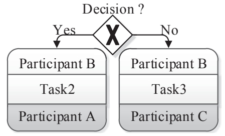
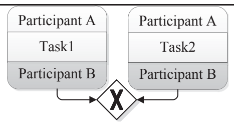
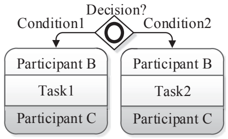
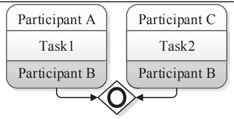
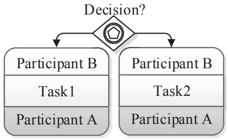
The `generate_ltl_code(`i`)` function is responsible for generating LTL formulas for each construct of a GCM. The pair of triple apostrophes (''') represents the string templates that are used for code generation based on Eclipse Xtend framework. However, a pair of guillemots (« and ») is used to represent the parametrised place-holders that will be bound to and substituted with the actual values extracted from the input model elements by the Xtend engine. The `generate_ltl_code(`i`)` function is not realized as a single function but rather a polymorphism of multiple functions. That is, depending on the type of the input interaction i, a particular function for generating LTL formulas for that interaction will be invoked. The LTL-based transformation rules for Parallel Join is presented in Algorithm 1. In particular, LTL formula for Parallel Join requires visited predecessors that are joined using the logical *AND* operator ("&") and offered to Parallel Join. The Parallel Join cannot execute until all incoming flows have been received. Table I summarises the constructs of choreography models along with their informal descriptions extracted from the BPMN 2.0 specification [6] and LTL-based transformation rules that constitutes the interactions between participants.

B. Generating SMV Descriptions from LCMi

This section concerns the generation of formal SMV descriptions from the local choreography models (*LCMi*). To define the interactions within a BPMN 2.0 collaboration diagram, 2-tuple $\langle participant\_name, task\_snd/task\_rec \rangle$ is used to represent the participant name and send task/receive task. The mapping of (*LCMi*) into SMV descriptions is attained using an extended version of the breadth-first search, as shown in Algorithm 2. Similar to GCM, three helper functions

[1] See https://eclipse.org/xtend

TABLE I: LTL-Based Transformation Rules for BPMN Global Choreography Model

BPMN Choreography	Modelling Notation	LTL-Based Transformation Rules
Sequence: (i) The sending action of a choreography task must exist before its receiving action. (ii) The initiator of a choreography task (excluding first activity) can not send a message to the receiver until it has received the prior message.	 Participant A Task1 Participant B Participant B Task2 Participant C	1)G (ParticipantA_Task1_Snd -> F (ParticipantB_Task1_Rec))& H (ParticipantB_Task1_Rec -> O (ParticipantA_Task1_Snd)) 2)F (ParticipantB_Task2_Snd)-> (! ParticipantB_Task2_Snd U (ParticipantB_Task1_Rec))
Parallel Fork: The execution of a Parallel Fork (AND-Split) leads to the parallel execution of subsequent choreography tasks. The initiators of all choreography tasks immediately following the Parallel Fork must be same as the common sender or receiver of choreography tasks preceding the gateway.	 Participant A Task1 Participant B Participant A Task2 Participant B	G (Fork -> F (ParticipantA_Task1_Snd & ParticipantA_Task2_Snd))& H ((ParticipantA_Task1_Snd & ParticipantA_Task2_Snd)-> O Fork)
Parallel Join: The concurrent execution of multiple interactions lead to the execution of a Parallel Join (AND-Join) gateway. However, all incoming branches have to be completed before the execution of a Parallel Join.	 Participant A Task1 Participant B Participant C Task2 Participant B	G ((ParticipantB_Task1_Rec & ParticipantB_Task2_Rec)-> F Join)& H ((Join)-> O (ParticipantB_Task1_Rec & ParticipantB_Task2_Rec))
Exclusive Decision: The execution of an Exclusive Decision (XOR-Split) is spawn in two or more branches, which branch is actually traversed depends on the evaluation of the guards on the outgoing flows.	 Decision ? Yes No Participant B Task2 Participant A Participant B Task3 Participant C	(ExclusiveDecision -> F ((ParticipantB_Task2_Snd & ! ParticipantB_Task3_Snd)| (! ParticipantB_Task2_Snd & ParticipantB_Task3_Snd)))
Exclusive Merge: The execution of one of the choreography receiving action among a set of alternative receiving actions will lead to the execution of an Exclusive Merge (XOR-Join).	 Participant A Task1 Participant B Participant A Task2 Participant B	(G (ParticipantB_Task1_Rec & ! ParticipantB_Task2_Rec)| (! ParticipantB_Task1_Rec & ParticipantB_Task2_Rec)-> F ExclusiveMerge)
Inclusive Decision: An Inclusive Decision gateway (OR-Split) represents the execution of one or more alternative branches. The traversal of branches depend on the evaluation of the guard conditions. In particular, all sequence flows with a true evaluation will be traversed.	 Decision? Condition1 Condition2 Participant B Task1 Participant C Participant B Task2 Participant C	G ((InclusiveDecision & Condition1)-> F (ParticipantB_Task1_Snd))| G ((InclusiveDecision & Condition2)-> F (ParticipantB_Task2_Snd))
Inclusive Merge: The alternative but also parallel execution of two or more active interactions lead to the execution of the Inclusive Merge gateway (OR-Join).	 Participant A Task1 Participant B Participant C Task2 Participant B	(G (ParticipantB_Task1_Rec | ParticipantB_Task2_Rec)-> F InclusiveMerge)
Event-Based: The execution of an Event-based gateway is spawn in two or more branches, which branch is actually traversed depends on a specific Event that occur. Usually, the receipt of a message or timeout determines the path that will be taken rather than the evaluation of the guards.	 Decision? Participant B Task1 Participant A Participant B Task2 Participant A	G (Event-basedgateway & rec_msg1) -> F (ParticipantB_Task1_Snd & (! ParticipantB_Task2_Snd))| G (Event-basedgateway & rec_msg2)-> F ((! ParticipantB_Task1_Snd)& ParticipantB_Task2_Snd)

are created, namely `get_events(`*el*`)`, `get_interaction(`*i*`)` and `generate_smv_code(`*i*`)`. The function `get_events(`*el*`)` returns a set of start events concerning the input *LCMi*. The function `get_interaction(`*i*`)` extracts all interactions such as choreography tasks, control nodes and connecting edges. In particular, given a certain interaction *i*, the outgoing interactions (within pool) can be attained using the function `get_interaction(`*i*`)`. An interaction *j* is called "outgoing interaction" of *i* if there is a sequence flow going from *i* to *j*. In a similar way, communication between two participants (pools) can be achieved. An interaction *j* is called "receiving interaction" of *i* if there is a message flow going from *i* to *j*. Thus, a set of receiving actions of choreography tasks and outgoing interactions of *i* can be achieved by following all of its message flows and outgoing sequence flows, respectively.

The function `generate_smv_code(`*i*`)` is responsible for generating the SMV description for each interaction within the *LCMi*. In particular, the aforementioned 2-tuple, control node or event will be represented by a boolean state variable in the section `VAR` and its corresponding state transitions will be described in the section `ASSIGN` by a combination of two functions given in NuSMV. The `init()` is used for assigning the initial state of a variable and `next()` is used for defining the transition to the next state. The function `next()` is often combined with the branching structure "`case/esac`" for selecting one of many possible choices. The state is initially set to `false`. However, if the incoming condition(s) are satisfied, it is changed to a `true` state (see Line 22 in Algorithm 2). The incoming condition(s) would be a guard expression and/or finishing of the preceding interaction(s). The interaction's state shall be switched back to `false` after finishing the execution (see Line 23 in Algorithm 2). Note

Algorithm 2 Generating SMV Descriptions from *LCMi*

```
procedure TRANSLATE($LCMi$)
    $Q \leftarrow \varnothing$                ▷ $Q$ is the queue of non-visited interactions
    $V \leftarrow \varnothing$                ▷ $V$ is the queue of visited interactions
    $Q \leftarrow Q \cup$ get_start_events($el$)
    for all $i \in Q$ do
        $V \leftarrow V \cup \{i\}$
        $Q \leftarrow Q \setminus \{i\}$
        generate_smv_code($i$)
        $I_{outgoing\_interactions} \leftarrow$ get_interaction($i$)
        $I_{receiving\_interactions} \leftarrow$ get_interaction($i$)
        for all $j \in I_{succeding\_interactions} | j \in I_{receiving\_interactions}$ do
            if ($j \notin V$) then
                $Q \leftarrow Q \cup \{j\}$
    extracts interaction information;
    binds input values and generates SMV descrip-
  tions using the following templates:
    '''
    VAR
        «i» : boolean;          ▷ State variable declaration
    ASSIGN
        init(«i») := «interaction-initial-state»
        next(«i») := case
            «incoming-condition(s)» : TRUE;
            «i» : FALSE;
        esac;
    '''
```

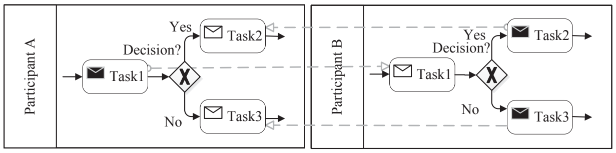
that the generate_smv_code(i) is not realised as a single function but rather a polymorphism of multiple functions. That is, depending on the type of the input interaction i, a particular function for generating SMV descriptions for that node type will be invoked. The subsequent sections discuss the rules for generating SMV descriptions for each node type that constitutes the individual function generate_smv_code(i).

1) Task, Fork, Join, End Event and Start Event: This section focuses on a set of elements that are triggered with respect to their incoming flows and are formalised rather similar in SMV. Listing 1 illustrates the translation of Task, Fork, Join and End Event into SMV descriptions based on the translation template shown in Algorithm 2. If an interaction has multiple incoming flows, the logical AND operator ("&") is used to represent the implicit "and-join" guard for all tokens passing through the incoming flows. Note that none Start Event is a special event that denotes the starting point of a BPMN model. It does not have any incoming flows. Thus, each none Start Event is represented by a boolean state variable whose initial state would be assigned as true.

```
VAR
   «interaction» : boolean;
ASSIGN
   init(«interaction») := FALSE;
   next(«interaction») := case
      «incoming_1» & «incoming_2» & ... & «incoming_n» : TRUE;
      «interaction» : FALSE;
   esac;
```

Listing 1: Generic Rules for Generation SMV Descriptions

2) Branching: The Exclusive Decision, Inclusive Decision, and Event-based gateways are the branching constructs in BPMN specification [6]. The execution of the Exclusive Decision will trigger one of the outgoing flows according to the corresponding guard conditions. The initiating participant of the messages that follow the gateway controls the decision. Figure 4 shows the rules for mapping an Exclusive Decision into SMV descriptions whose guard conditions is abstracted as boolean variables (Line 7). We introduce a temporary variable named post_decision_i in which i is an incrementally generated number for exclusively choosing one of many alternative sequence flows. The variable post_decision_i has an enumerated type which includes a normal state "undetermined" and the values corresponding to the sequence flows (Line 9). The choice among alternative sequence flows is made using a "case/esac" construct (Line 16–20). In case none of the guard conditions is true, the state transitions defined in Figure 4 will be stuck. This is precise, but undesired behaviour. To avoid this stuck a "Default Condition" for one of the outgoing sequence flows can be used. The default condition is a complement of other guard conditions and will be chosen when all other conditions turn out to be false.

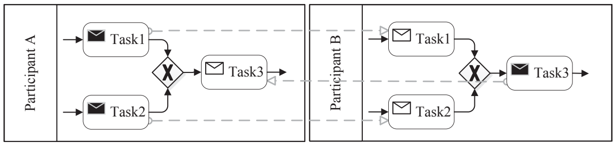


```
1  VAR
2     «ExclusiveDecision»: boolean;
3     «ParticipantB_Task1_Rec» : boolean;
4     ...
5     «ParticipantB_Task3_Snd» : boolean;
6     -- abstraction of boolean expressions
7     «guard_1» : boolean;
8     -- temporary variable
9     «post_decision_i» : {undetermined, «out_yes», «out_no»};
10 ASSIGN
11    init(«ExclusiveDecision») := FALSE;
12    next(«ExclusiveDecision») := case
13       «ExclusiveDecision» : FALSE;
14    esac;
15 ... -- the initializations of guards are omitted
16    init(«post_decision_i») := undetermined;
17    next(«post_decision_i») := case
18       «ExclusiveDecision» & «guard_1» : «out_yes»;
19       «ExclusiveDecision» & «guard_2» : «out_no»;
20       TRUE : undetermined;
21    esac;
22    init(«ParticipantB_Task2_Snd») := FALSE;
23    next(«ParticipantB_Task2_Snd») := case
24       «post_decision_i» = «guard_1» : TRUE;
25       «ParticipantB_Task2_Snd» : FALSE;
26    esac;
27    init(«ParticipantB_Task3_Snd») := FALSE;
28    next(«ParticipantB_Task3_Snd») := case
29       «post_decision_i» = «guard_2» : TRUE;
30       «ParticipantB_Task3_Snd» : FALSE;
31    esac;
```

Fig. 4: SMV Generation Rules for Exclusive Decision

An Inclusive Decision represents the execution of any number of branches instead of one or all. The translation rules of Inclusive Decision are similar to Exclusive Decision; however, a "true" evaluation of one guard condition does not exclude the evaluation of other guard conditions. All sequence flows with a "true" evaluation will be traversed by a token. The Event-based gateway represents an alternative branching point where the decision is made by two or more events; for instance, the choice for an outgoing sequence flow is made when an event will occur on the particular outgoing flow. When execution of process arrives at the particular point, the execution stops until either the message event or the timer event occurs. However, the occurrence of first event will immediately continue its outgoing sequence flow by disabling the other paths. In this case, a boolean variable *wait_event* is used to indicate that the outgoing flows can not be proceed until an event is occurred. The SMV descriptions of `Inclusive Decision` and `Event-based` gateways can be derived from the `Exclusive Decision`. The complete translation details cannot be presented due to space reasons and similar technical details.

3) Exclusive Merge and Inclusive Merge: An `Exclusive Merge` brings together multiple alternative interactions and exclusively accepts one among them. If an `Exclusive Merge` has two incoming interactions, a straightforward naive encoding strategy is to use the xor operator “*ParticipantB_Task*1*_Rec* `xor` *ParticipantB_Task*2*_Rec*” to express the incoming guard condition of an `Exclusive Merge`. However, this strategy cannot be effectively generalised for `Exclusive Merge` that has more than two incoming sequence flows because the operator “`xor`” with n operands ($n \geq 3$) yields `true` not only when one of its operands is `true` but also when the odd numbers of the operands are `true` [8]. `Exclusive Merge` is therefore implemented by its equivalent but longer form “$(ParticipantB\_Task1\_Rec \wedge \neg ParticipantB\_Task2\_Rec) \vee (\neg ParticipantB\_Task1\_Rec \wedge ParticipantB\_Task2\_Rec)$”.



```
1  VAR
2     «ParticipantB_Task1_Rec» : boolean;
3     «ExclusiveMerge» : boolean;
4     «merge_flag_i» : {undetermined, «in_t1», «in_t2»}
5  ASSIGN
6     ... -- the initializations and transitions of tasks are omitted
7     init(«merge_flag_i») := undetermined;
8     next(«merge_flag_i»):= case
9        («merge_flag_i» = undetermined) & («ParticipantB_Task1_Rec»
              | «ParticipantB_Task2_Rec»): {«in_t1», «in_t2»};
10       TRUE : undetermined;
11    esac;
12    init(«ExclusiveMerge») := FALSE;
13    next(«ExclusiveMerge») := case
14       «ParticipantB_Task1_Rec» & ! «ParticipantB_Task2_Rec» :
              TRUE;
15       ! «ParticipantB_Task1_Rec» & «ParticipantB_Task2_Rec» :
              TRUE;
16       «merge_flag_i» = «in_t1» | «merge_flag_i» = «in_t2» : TRUE
              ;
17       «ExclusiveMerge» : FALSE;
18    esac;
```

Fig. 5: SMV Generation Rules for Exclusive Merge

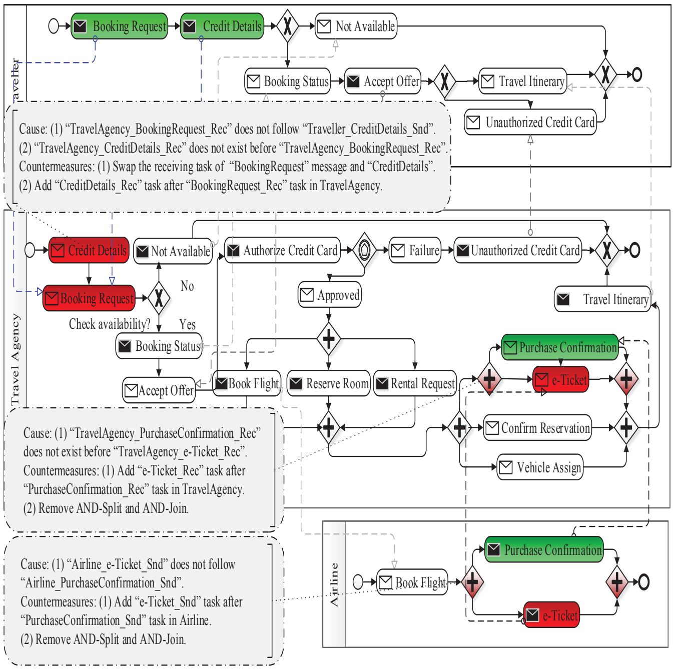
In order to avoid name conflicts, a temporary variable `merge_flag_i` is introduced for each `Exclusive Merge`, where *i* represents an incrementally generated number to avoid name conflicts, as shown in Figure 5. This temporary variable has an enumerated type that comprises “`undetermined`” and “`in_t`x”. The former denotes the normal state; whereas the latter represent the state values that correspond to the incoming sequence flows ($x = 1,...,n$). As we see in Line 9, one branch will be non-deterministically and exclusively selected from the activated branches. That is, in order to verify in case some k ($k \leq n$) incoming interactions are simultaneously activated, NuSMV will bind `merge_flag_i` to a certain value “`in_t`x” in the first place, to “`undetermined`” in the next transition, then to another value “`in_t`y” in the subsequent transition, and so forth. In combination with the branching construct “`case/esac`” (Line 12–18), we can see that the `Exclusive Merge` is activated if and only if either one of the incoming interactions is `true` or the variable `merge_flag_i` is assigned to a state value “`in_t`x”, where $x = 1,...,n$.

An `Inclusive Merge` has similar semantics to `Exclusive Merge`; however, it brings together not only multiple alternatives but also parallel interactions and accepts one or more among them. In the case of `Inclusive Merge`, we use the logical `OR` operator (“|”) to express the incoming guard condition instead of `xor` operator.

C. Containment Checking and Dealing with Violations

This section is devoted to the identification of containment problems and their resolutions. The containment violations may occur due to a variety of reasons, such as (i) missing participant or interaction – a participant or interactions exist in the global model may not exist in the local choreography models; (ii) misplacement of elements – the local choreography model contains interactions with participant specified in the global choreography model but with different structure. To alleviate containment checking problems, an efficient analysis of the generated counterexample is supported in the proposed approach. The automated counterexample analysis not only detects the actual causes of the unsatisfied containment relationship but also provides appropriate guidelines to resolve the particular violations. Therefore, the output trace file is scrutinised and parsed to determine the unsatisfied LTL formulas. The extracted formulas and SMV descriptions together with LTL-based transformation rules are traversed to find out why the elements of the global choreography model are not matched with their corresponding local choreography counterparts. This is performed in two steps. Firstly, the missing element cause (either one, multiple, or all elements could be missing) is detected and the countermeasure (i.e., insert the missing element at a particular position in the model) is suggested. Secondly, the sequence of elements from the SMV descriptions is scrutinised and corresponding elements (e.g., tasks, gateways and so on) causing the violation of the LTL formulas are located (i.e., misplacement of elements) and relevant countermeasures (i.e., add, delete or replace the element after or before the particular element) are suggested.

The counterexample analysis results are presented in Figure 6. The gray boxes display the actual causes and potential countermeasures of the unsatisfied formulas. Furthermore, the elements responsible for causing the containment violation are highlighted in red; whereas the elements that satisfied the rule are highlighted in green. In this case, the containment relationship is not satisfied due to the violation of sequential rules. The receiving rule for the `RequestBooking` and `CreditDetails` messages is violated because *traveller* invokes the *travel agency* by sending `RequestBooking` message before `CreditDetails` message; however, *travel agency* receives the `CreditDetails` message before the `RequestBooking` message. This implies that either receiving event of one or both tasks are misplaced. It can be resolved by putting the `CreditDetails` receive task after the `RequestBooking` receive task in the local choreography model of *travel agency*.



Cause: (1) “TravelAgency_BookingRequest_Rec” does not follow “Traveller_CreditDetails_Snd”. (2) “TravelAgency_CreditDetails_Rec” does not exist before “TravelAgency_BookingRequest_Rec”. Countermeasures: (1) Swap the receiving task of “BookingRequest” message and “CreditDetails”. (2) Add “CreditDetails_Rec” task after “BookingRequest_Rec” task in TravelAgency.

Cause: (1) “TravelAgency_PurchaseConfirmation_Rec” does not exist before “TravelAgency_e-Ticket_Rec”. Countermeasures: (1) Add “e-Ticket_Rec” task after “PurchaseConfirmation_Rec” task in TravelAgency. (2) Remove AND-Split and AND-Join.

Cause: (1) “Airline_e-Ticket_Snd” does not follow “Airline_PurchaseConfirmation_Snd”. Countermeasures: (1) Add “e-Ticket_Snd” task after “PurchaseConfirmation_Snd” task in Airline. (2) Remove AND-Split and AND-Join.

Fig. 6: Visual Support for Understanding and Resolving Containment Violations

Similarly, the primary root causes of other violations are due to the execution of `PurchaseConfirmation` and `e-Ticket` messages in parallel order instead of sequential order. These violations can be resolved by deleting forks and joins, and putting the `e-Ticket` message after the `PurchaseConfirmation` message in the local choreography models of *travel agency* and *airline*. Once the causes are located, they are eliminated by updating the responsible elements of the choreography models and rerunning the containment checking process yielded no further violations. Without the counterexample analysis, users would have to study and investigate the syntax and semantics of the trace file in order to determine the relationship between the execution traces and the service choreography models, and then locate the corresponding violation within models, meaning that the complex matching between the variables and states in the counterexample and the elements of the choreography models is performed manually. This is especially cumbersome for those having limited knowledge of the underlying formal techniques.

IV. Evaluation

We implement containment checking approach and conduct a preliminary evaluation of its performance. The main idea is to validate whether the proposed approach performs reasonably for typical models used in industry on typical workstations used by developers. The workstation used for the performance evaluation is running under Windows 8 on a 2.6 GHz i5 processor with 8GB of memory using NuSMV 2.5.4. The evaluation is conducted through three behaviour models of different sizes and complexity that are taken from our previous industry projects. One of them is the *Travel Booking* (TB) mentioned in the previous section. The other two are *Automated Teller Machine* (ATM) and *Order Processing* (OP). We omit the details of OP and ATM scenarios due to space limitations. Table II shows the complexity of the input BPMN model (GCM = global choreography model, LCMi = local choreography models) with respect to their elements including tasks, gateways, and edges (sequence and message flows).

TABLE II: Model Size and Translation Time

Input size	**OP**		**TB**		**ATM**	
	GCM	**LCMi**	**GCM**	**LCMi**	**GCM**	**LCMi**
Gateways	7	19	7	28	8	19
Interactions	16	32	17	34	22	44
Edges	25	72	27	83	32	89
Total Elements	48	123	51	145	62	152
Model Loading (ms)	2.351±0. 59	3.387±0. 65	3.215±0. 39	4.984±0.27	3.278±0. 14	5.620±0. 87
Translation Time (ms)	0.315±0.19	0.514±0. 98	0.541±0.05	0.784±0.44	0.596±0.22	0.861±0.07

Table III shows the total execution time of three models, reachable states and violated formulas. The evaluation results indicate that the containment checking time spent by NuSMV for the TB process is longer than the ATM and OP. This is because NuSMV found violations between the formal descriptions of the LCMi and LTL formulas of the GCM and thus NuSMV needed to generate a counterexample for violated LTL formula. The evaluation results demonstrate that our approach efficiently translates service choreography models into formal descriptions and consistency constraints for supporting containment checking. In particular, all realistic scenarios are handled in a total time around a second which is quite reasonable for practical purposes. Our analysis and evaluation results based on the aforementioned use case scenarios show the feasibility of our approach for larger systems.

TABLE III: Performance Evaluation Results

Containment checking	**OP**	**TB**	**ATM**
Verification Time (ms)	265.0±11.952	816.25±7.440	463.75±13.025
Total Time (ms)	271.567	825.744	474.607
Violated Formulas	0 out of 34	4 out of 38	0 out of 47
Reachable States	5 (2ˆ2.32193)	1.87027e+015 (2^50.7322)	128 (2ˆ7)
Total States	2.15163e+023(2ˆ77.5098)	7.3584e+039 (2^132.435)	1.42772e+029(2ˆ96.8496)

V. Related Work

Zaha et al. [1] propose the algorithms for generating local models (i.e., provider behaviour) from global models and for verifying the local enforceability of global models. Yu et al. [9] propose an approach for the specification of properties called PROPOLS and for verification of BPEL schemas. The approach first translated the BPEL schemas and PROPOLS into Finite State Automatas (FSAs), then compares these FSAs. However, the approach does not deal with the service choreographies. Kwantes et al. [2] present the translation of the

BPMN collaboration diagram into an LTL formula to check conformance with local workflows as BPMN process diagrams using GROOVE tool. However, the translation has been done manually.

Poizat and Salaün [10] introduce the LOTOS NT process algebra formalism for BPMN choreographies to validate the realizability between models using the CADP state space exploration tools. In particular, the interactions produced by the global choreography model and communicating peer processes are compared. Fu et al. [11] present a formal specification, verification, and analysis tool for web service compositions based on guarded automata (GA). BPEL specifications are translated to GA and then mapped to Promela, the input language of the SPIN model checker. Solaiman et al. [12] developed a BPMNverifier tool that automatically converts BPMN choreography models into Promela. However, the LTL properties are manually created or otherwise retrieved for the generated Promela models; they are stored in a repository. These approaches require a considerable amount of knowledge of temporal logics properties.

In the course of our earlier research, we have investigated the containment checking problem for activity diagrams [13] and sequence diagrams [14]. In addition to the model checking based techniques, a lightweight graph-based approach has been proposed that verifies missing nodes, missing transitive links, and missing cycles [15]. This research focuses on the containment relationship between global and local choreography models, which has not been considered in the literature. The proposed approach not only provides formalisation for automated transformation of global and local choreography models into consistency constraints and formal descriptions, but also gives more informative and comprehensive feedbacks to developers/architects for identifying the causes of containment violations and their resolutions.

VI. Conclusion

Motivated by the need to support the containment checking in service choreographies, we introduced a set of transformation rules to facilitate the automated transformation of global and local choreography models into LTL constraints and SMV descriptions, respectively. This provides efficient means for automated generation of consistency constraints and formal descriptions for large and complex choreography models. The results produced by the model checkers (i.e., counterexamples) are rather cryptic and verbose, and thus, tracking the entire evidence is difficult for architects/developers. In order to mitigate the need for strong background of formal techniques, the counterexample analysis mechanism is integrated that provides more informative and comprehensive feedbacks to the stakeholders for identification of containment problems and their resolutions. To illustrate the applicability of the proposed approach, we realized use case scenarios of ATM machine, travel booking and order processing systems; the performance evaluation is also carried out in particular cases. By analysing the evaluation results we found that our approach efficiently translates choreography models into formal specifications and works well for larger realistic scenarios. In the future, we plan to conduct controlled experiments with participants from industry and academia to empirically validate whether the proposed approach significantly supports human analysts in identification and resolution of containment inconsistencies.

Acknowledgment

This work is supported by the Wiener Wissenschafts-, Forschungs- und Technologiefonds (WWTF), Grant No. ICT12-001 and University of Vienna.

References

[1] J. M. Zaha, M. Dumas, A. t. Hofstede, A. Barros, and G. Decker, “Service interaction modeling: Bridging global and local views,” in *Proceedings of the 10th IEEE International Enterprise Distributed Object Computing Conference*, ser. EDOC ’06, Hong Kong, China, 2006, pp. 45–55.

[2] P. M. Kwantes, P. V. Gorp, J. Kleijn, and A. Rensink, “Towards compliance verification between global and local process models,” in *Proceedings of the 8th International Conference on Graph Transformation*, ser. STAF ’15, L’Aquila, Italy, 2015, pp. 221–236.

[3] F. U. Muram, H. Tran, and U. Zdun, “Counterexample analysis for supporting containment checking of business process models,” in *Business Process Management Workshops - BPM 2015, 13th International Workshops, Innsbruck, Austria, August 31 - September 3, 2015, Revised Papers*, 2015, pp. 515–528.

[4] A. Pnueli, “The temporal logic of programs,” in *18th Annual Symposium on Foundations of Computer Science*, ser. SFCS ’77. IEEE Computer Society, 1977, pp. 46–57.

[5] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri, “Nusmv: A new symbolic model verifier,” in *Proceedings of the 11th International Conference on Computer Aided Verification*, ser. CAV ’99, Trento, Italy, 1999, pp. 495–499.

[6] Object Management Group (OMG), “Business Process Model and Notation (BPMN) Version 2.0.” http://www.omg.org/spec/BPMN/2.0, last accessed: May 8, 2017.

[7] K. Y. Rozier, “Survey: Linear temporal logic symbolic model checking,” *Comput. Sci. Rev.*, vol. 5, no. 2, pp. 163–203, May 2011.

[8] F. Pelletier, “Ternary Exclusive Or,” *Logic Journal of the Igpl*, vol. 16, no. 1, pp. 75–83, 2008.

[9] J. Yu, T. P. Manh, J. Han, Y. Jin, Y. Han, and J. Wang, “Pattern based property specification and verification for service composition,” in *Proceedings of the 7th International Conference on Web Information Systems*, ser. WISE’06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 156–168.

[10] P. Poizat and G. Salaün, “Checking the realizability of bpmn 2.0 choreographies,” in *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, ser. SAC ’12, Trento, Italy, 2012, pp. 1927–1934.

[11] X. Fu, T. Bultan, and J. Su, “WSAT: A tool for formal analysis of web services,” in *Proceedings of the 16th International Conference on Computer Aided Verification*, ser. CAV ’04, Boston, MA, USA,, 2004, pp. 510–514.

[12] E. Solaiman, W. Sun, and C. Molina-Jimenez, “A tool for the automatic verification of bpmn choreographies,” in *Proceedings of the 2015 IEEE International Conference on Services Computing*, ser. SCC ’15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 728–735.

[13] F. U. Muram, H. Tran, and U. Zdun, “Automated Mapping of UML Activity Diagrams to Formal Specifications for Supporting Containment Checking,” in *11th Int’l Workshop on Formal Engineering approaches to Software Components and Architectures (FESCA)*, Grenoble, France, Apr. 2014, pp. 93–107.

[14] ——, “A model checking based approach for containment checking of uml sequence diagrams,” in *23rd Asia-Pacific Software Engineering Conference (APSEC).* Hamilton, New Zealand: IEEE Computer Society, 2016.

[15] H. Tran, F. Ul Muram, and U. Zdun, “A graph-based approach for containment checking of behavior models of software systems,” in *Proceedings of the 2015 IEEE 19th International Enterprise Distributed Object Computing Conference*, ser. EDOC ’15, Adelaide, SA, Australia, 2015, pp. 84–93.