A LOTOS NT Library for Modelisation, Analysis, and Validation of Distributed Systems

Alexandre Dumont
alexandre.dumont@ensimag.imag.fr
Introduction to Laboratory Research 2012
INRIA - ENSIMAG
Tutor : Gwen Salaün (INRIA - Grenoble INP)

***Abstract*—Since software systems are designed more and more distributed and concurrent, modeling, analysis and validation of interactions among their components has become a capital concern. In several application domains, the components inside a distributed system interact with each other using message-based communication. Choreographies consist in the specification of these communication contracts among a set of services from a global point of view. They are the basis for the further development steps and their analysis may find out issues (*e.g.* deadlocks or requirement violations) which can induce delays or additional costs if detected lately. In this work, carried out in the *INRIA Convecs team*, we propose a framework for modelisation, analysis and validation of choreography specification. Therefore, we present first the intermediate format describing choreographies properly while accepting various choreography specification languages as input. Second, we focus on the LOTOS NT translation of the processes used in the choreography. Last, we present the connectivity to the CADP toolbox and our verification package to check properties (*e.g.* synchronizability, realizability, conformance...) on choreographies.**

I. Introduction

Software systems are nowadays becoming more and more distributed. They are no longer built as stand-alone programs, but rather as the parallel assembly of various collaborating entities. Using such a composition, software systems benefit more from the modern processors computing power and thus run more efficiently. In several application domains, the interaction mechanism among the components inside a distributed system relies on message-based communications. A *choreography* defines these interactions among a set of services from a global point of view [2]. In the design process of a distributed system, it is necessary to provide first a choreography which specifies how the entities behave. The analysis of this contract is capital for the further development steps because it may reveal design errors which could induce additional costs if discovered lately. To this day, only a few tools have been developed for choreography checking, and they mainly focus on realizability property [2,5]. The purpose of our work is to provide a framework for choreography analysis and verification.

Several formalisms are available to specify choreographies, such as collaboration diagrams, WS-CDL (*Web Services Choreography Description Language*), BPEL (*Business Process Execution Language*), Chor calculus [3] or BPMN 2.0 (*Business Process Modeling Notation*) [1,7]. In the following, we focus on Chor calculus and BPMN 2.0 to represent choreographies, and we provide a short description of these modeling languages in the next section.

A first work has made the realizability validation possible for BPMN 2.0 choreographies [1] and proposes an internal model to directly encode BPMN 2.0 specification into LOTOS NT processes. The developed tools also provide a verification package - using SVL scripts - fully connected to the CADP toolbox (*Construction and Analysis of Distributed Processes* - http://cadp.inria.fr) and return readable feedbacks to the designer. The framework we propose draws inspiration from this previous work. Its motivations are to accept various choreography description languages as input and to perform any kind of formal analysis on these specifications. The maintenance, development or integration operations performed on the framework get consequently easier and simpler due to its wide modularity and extensibility (see fig. 1).

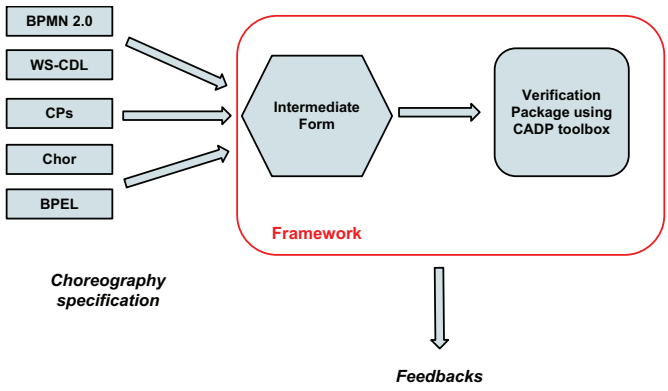


Fig. 1: *Framework - simple overview*

More precisely, the framework structure we have implemented is divided into three parts (see fig. 2) :

- An XML based intermediate format for choreography description accepting several languages as input. This representation is a connection interface for choreography description languages such as BPMN 2.0 or Chor calculus.
- A Python internal model which makes the transition between the XML intermediate format and the LOTOS NT processes. It includes primitives for LOTOS NT code

and SVL scripts generation.

- A verification library connected to the CADP toolbox, automating some analysis and verification of interesting properties such as realizability, synchronizability and deadlock freeness...

The personal contributions to this framework result in the intermediate format design and changes in the connection with the internal model. The connection to the verification package and its improvements have been achieved by *Inria Convecs team* members.

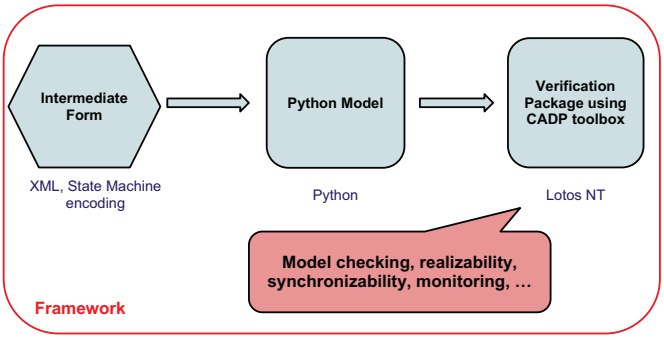


Fig. 2: *Framework - detailed overview*

The organization of the rest of this paper is as follows. Section II gives a short description of two choreography specification notations, respectively BPMN 2.0 and Chor calculus. In section III, we present our intermediate format and the way it is connected to the internal Python model, described in section IV. Last, section V details the verification package we implemented and section VII adds some concluding remarks to the report.

II. Choreography Description Languages

In this section, we present two formalisms available to specify choreographies. First, we focus on BPMN 2.0 notation, then we deal with the Chor calculus which is a more theoretical specification language for choreographies. Semantically, a choreography is specified as a state-machine, and the following notations are similar to *Labelled Transition Systems* (LTS for short) including states with particular properties depending on their type.

BPMN 2.0: BPMN 2.0 [7] is a standard for business process modeling that provides a graphical notation to specify business processes in a Business Process Diagram. As a comparison, the obtained graphs are very similar to UML activity diagrams. BPMN 2.0 offers a wide range of activity operators and so expresses many processes out of the choreography context. Therefore, only a few operators match choreography specification. The basic building block of BPMN 2.0 (BPMN for short) Choreography Diagrams is a one-way or two-way interaction between peers (see fig. 3). The peer which initiates the interaction is represented in a white band whereas the other is in a gray filled band. With these interactions come message flows described using a white envelope and a black one, respectively for the initiating and return messages. One can notice that the return message is optional. Sequence flow operators allow to define an execution order for a set of several interactions.

Concretely, among all the descriptors we consider, one can find :

- initial and final states, respectively initiating and ending the choreography.
- activity states, corresponding to the emission of a message between two participants, namely the sender and the receiver (graphically represented by an envelope).
- exclusive gateways, matching choice behavior between two threads.
- inclusive gateways, meaning an inclusive choice between one or several different execution threads.
- parallel gateways, selecting all the execution threads.
- sequence flows, linking two of the previous states in the diagram.

The graphical notation for these operators is given in fig. 3.

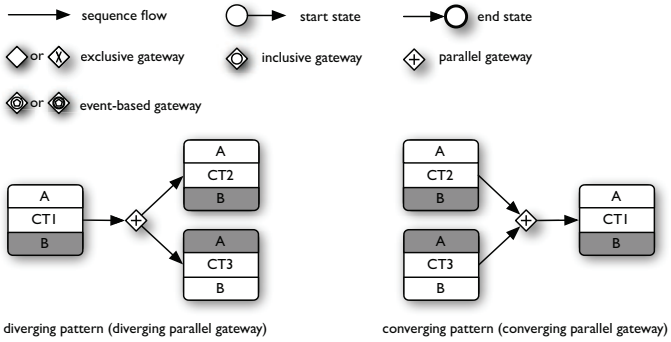


Fig. 3: *BPMN 2.0 - Choreography operators [9]*

Chor calculus: This notation provides choreography specification in a more formal way through an abstract grammar precised in fig 4. We find similar operators such as choice or parallel composition structure, communication $c^{[i,j]}$ between peers i and j ... One can notice that there is an additional operator, named a^i - activity in role R^i - which refers to an internal action performed by the peer i. Our work focuses on message transmission between peers, thus this operator is not supported yet in our framework.

$$
\begin{array}{lll}
A & ::= BA & \text{(basic activities)} \\
 & \mid A;A & \text{(sequential composition)} \\
 & \mid A \sqcap A & \text{(choice)} \\
 & \mid A \parallel A & \text{(parallel composition)} \\
BA & ::= \mathsf{skip} & \text{(no action)} \\
 & \mid a^i & \text{(activity in role } R^i\text{)} \\
 & \mid c^{[i,j]} & \text{(communication)}
\end{array}
$$

Fig. 4: *Syntax of Chor [3]*

To illustrate this formalism, let us take the example of the following choreography specification using Chor notation :

$$Choreo \triangleq (\ c^{[1,2]} \ ; \ c^{[2,1]} \ ) \ || \ c^{[3,4]}$$

where 1 .. 4 refer to the four participants. It expresses three communications between four peers, and the parallel composition operator || denotes that the threads ($c^{[1,2]}$; $c^{[2,1]}$) and $c^{[3,4]}$ run concurrently. More precisely, the first thread is equivalent to the emission of a message from peer 1 to peer 2, followed by a return message from peer 2 to peer 1. The other thread produces a concurrent behavior consisting in the communication from peer 3 to peer 4. In addition to this set of operators, [3] introduces the notion of *dominated choice*. It has the same semantics as the choice construct - *i.e.* it selects one and only one execution thread between a set of several ones - and precises a peer which makes this choice. This construct solves some realizabilty problems when the choreography is implemented in a distributed system (explained in section V). One can notice that such a representation is similar to a LTS and fully corresponds to choreography specification.

III. Intermediate Format Representation

The framework we have implemented for choreography verification relies on a XML based intermediate format. Such a model presents several advantages. First, it accepts a large number of choreography description languages as input. Therefore it serves as a modular connection interface for choreography specification notations. Second, our format is easily extensible with new choreography constructs - because it is based on XML - and is expressive enough to fully describe peer interactions inside a choreography. Last, it allows to perform formal analysis on choreographies using the CADP toolbox, and makes possible to use any other formal verification tool on condition that a connection to those tools is provided.

XML to encode choreographies: Semantically, the communication contract among the components of the system is specified as a state machine. The XML structure of our intermediate format preserves this state machine pattern. We first encode the list of all participants, *i.e.* the peers involved in the choreography, then the set of messages transmitted between peers, *i.e.* the alphabet, and last we express the state machine as specified in the choreography. A state in the machine encodes either an interaction, either a choreography operator such as choice, parallel splits or merges... Within each state are encoded zero, one, or several successors which refer to the transitions in the corresponding LTS. So as to express in a better way the structure of our state machine intermediate format, we provide the class diagram given in figure 6 (see next page).

More precisely, all the choreography operators are represented as states in the state machine. We introduce an abstract class *State* which specifies the most common attributes for each state in the choreography, *e.g.* its identifier and successors list. From this basic state definition we declare the *InitialState* and *FinalState* classes. We enforce that their successors list are bounded, respectively of size 1 and 0 - it is clear that a final state has no successors.

In addition, the *Interaction* class provides the identifier of the message exchanged between two peers. Data about sending and receiving peers, or about the message content are located in the *alphabet* section of the XML encoding.

```
<choreography>
  <choreoID>Running Example</choreoID>

  <participants>
    <peer>
        <peerID>Client</peerID>
    </peer>
    <peer>
        <peerID>Server</peerID>
    </peer>
  </participants>

  <alphabet>
    <message>
      <msgID>m1</msgID>
      <sender>Client</sender>
      <receiver>Server</receiver>
      <messageContent>...</messageContent>
    </message>
  </alphabet>

  <stateMachine>
    <initial>
      <stateID>s0</stateID>
      <successors>s1</successors>
    </initial>

    <interaction>
      <stateID>s1</stateID>
      <successors>s2</successors>
      <msgID>m1</msgID>
    </interaction>

    <final>
      <stateID>s2</stateID>
    </final>
  </stateMachine>
</choreography>
```

Fig. 5: *Choreography encoding*

We also provide two abstract class to describe splitting and merging gateways. Let us focus on splits first. The abstract class *Selection* is implemented by the following concrete classes :

- *Choice* means an exclusive choice between two or more successors.
- *DominatedChoice* has the same structure as *ChoiceState*, and one must define the peer that effectively make the choice.
- *SubsetSelect* specifies an inclusive choice within a set of successors. When this class is instantiated, one must precise a default path in the graph if no choice is made.

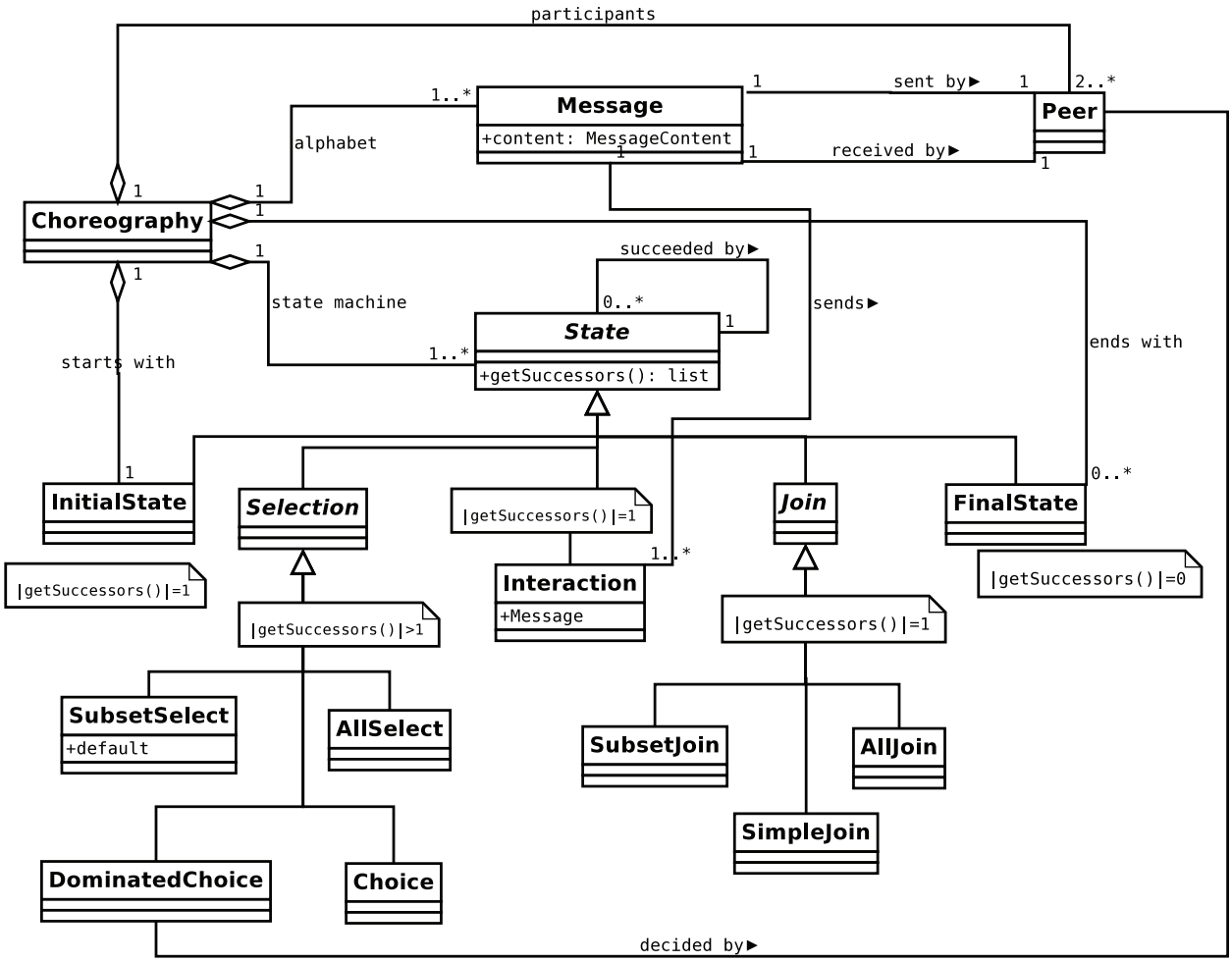


Fig. 6: *XML Format Class Diagram for Choreographies [9]*

- *AllSelect* corresponds to a parallel selection of all successors. It triggers concurrent executions in the distributed system.

Last, the *Join* class represents merging gateways and is concretely implemented by *AllJoin*, *SimpleJoin* and *SubsetJoin* corresponding to the associated splitting gateways. Fig. 5 provides a short example of a choreography expressed in our intermediate format.

***XSD checking**:* In complement to the XML classes, we provide a XSD document to specify the internal organization of the intermediate format. XSD - *XML Schema Definition* - defines a set of rules to which an XML document must conform in order to be considered as valid according to that schema. We introduce *complex types* to precise peer, message and choreography operator structures. For instance, we enforce that the choreography description contains, in the following order : a choreography identifier, a list of peer descriptions, a list of exchanged messages and the state machine encoding.

Using XSD document, we add constraints on the intermediate format, *e.g.* the participants list contains at least two peers (and the maximum number of elements is declared as unbounded) or the structure of a *message* element must contain the sequence of an identifier, the sender and receiver tags, and then the actual content of the message. We give a short excerpt of the XSD specification in fig. 7 for illustration purposes.

```
<xs:simpleType name="id">
  <xs:restriction base="xs:string" />
</xs:simpleType>

<xs:complexType name="peer">
  <xs:sequence>
    <xs:element name="peerID"
                type="id" />
  </xs:sequence>
</xs:complexType>

<xs:complexType name="peerList">
  <xs:sequence minOccurs="2"
               maxOccurs="unbounded">
    <xs:element name="peer"
                type="peer" />
  </xs:sequence>
</xs:complexType>
```

Fig. 7: *XSD specification - Participants*

Basic types are represented using a *xs:simpleType* tag and more complex types are described within a *xs:complexType* tag which contains a sequence of elements, each associated to a previously declared type. The framework we propose takes first the XSD document to check whether or not a choreography expressed in XML matches our intermediate format.

IV. Internal Model - from XML to LOTOS NT

In this section, we present the second part of the framework which interfaces the XML intermediate format representation of choreographies with the verification package. This step of modelisation consists in transposing the XML-based choreography into LOTOS NT processes. To reach this goal, we use an internal model written in Python language. This model offers primitives for process generation, and its structure is very similar to the class diagram presented in fig. 6. In order to extract relevant information from the XML document, we use the python library *PyXB* which provides a XML parsing tool. Furthermore, this tool uses the XSD specification document to first check the choreography encoding conformance to the intermediate format, and second to generate classes matching the XSD element tags. Let us illustrate the interaction operator, considering the following basic XML block described via XSD notation :

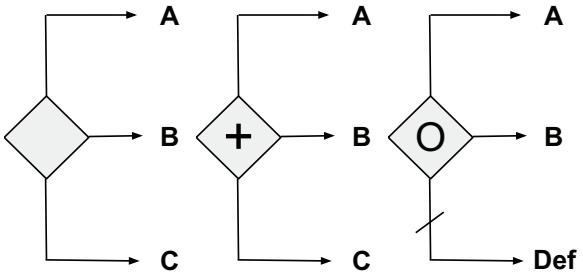
```
<xs:complexType name="interactionState">
  <xs:complexContent>
    <xs:extension base="oneSuccState">
      <xs:sequence>
        <xs:element name="msgID"
                    type="id" />
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Fig. 8: *The excerpt of the interaction state*

We assume that the *complexType* entity *oneSuccState* contains two attributes, respectively its name (the *stateID* attribute) and its successors list, bounded to size one. The resulting class *InteractionState* in the Python model contains the three following attributes : *stateID* and *successors* are inherited from the super class ; the last one is named *msgID* and contains the identifier of a message. The data structures built by PyXB parser to manage XML documents make easier the transposition of the intermediate format into Python internal data models.

***Encoding into LOTOS NT**:* Once we have an internal data model for representing communication contracts among a set of peers, we need to provide a connection to CADP tools in order to enable the verification of choreographies. The CADP toolbox contains powerful model and equivalence checking techniques. These tools are particularly convenient to check properties in distributed services such as synchronizability, realizability, conformance or deadlock detection... (see section V for further details).

As for making this connection effective, we chose LOTOS NT which is one of the CADP input specification languages. LNT is a process algebra, *i.e.* a theoretical approach for formally modelling concurrent and/or distributed systems. CADP toolbox offers LNT-related tools for the specification of processes and the formulation of statements about them. LNT is an expressive enough process algebra for accepting all the constructs introduced in our intermediate format. The formal analysis we perform in our framework consists in the verification of these statements, therefore we implement primitives in our internal data representation to generate LNT code associated to each state in the automaton described in the choreography. When encoding the state machine specification into LNT, we preserve the state machine representation used in our intermediate format. Each state is encoded in the machine as an LNT process. This process corresponds to the behaviour of the corresponding state, plus a call to the process encoding the successor state. The possible messages exchanged during the execution of the process are specified as parameters in the LNT encoding (represented within `[...]` structure in the examples below). In case of particular choreography constructs like selection operators, it may exist several process calls if there are several successors.



(a) *Choice* (b) *Parallel* (c) *Inclusive*

Fig. 9: *Selection Constructs*

We provide in fig. 9 some examples about choice operators and parallel gateways, both translated from graphical notation to concrete LNT code. Example *a)* denotes the choice construct whereas example *b)* refers to parallel selection operator. The last gateway, presented in *c)* represents the inclusive gateway state. One can notice that the *SubsetSelect* construct includes a default path, in case of no choice is made.

```
process Example_a [...] is
    select
        proc_A[...]
        [] proc_B[...]
        [] proc_C[...]
    end select
end process
```

Fig. 10: *Choice - LNT code*

Each construct is associated to a particular behaviour, and therefore corresponds to a specific LNT operator. More precisely, the LNT code generated for operator *a)* is given in fig. 10.

Semantically, the LNT operator `[]` means an exclusive selection among a set of different optional paths - respectively `proc_A`, `proc_B` and `proc_C` in the running example. These three paths are each associated to a state in the choreography specification, therefore they are encoded as LNT processes. The example *b)* introduces the parallel composition operator - `||` - which is an other construct used to describe concurrent processes. Its semantics are to activate all of the outgoing branches simultaneously, *i.e.* the three processes run concurrently.

```
process Example_b [...] is
    select
        proc_A[...]
        || proc_B[...]
        || proc_C[...]
    end select
end process
```

Fig. 11: *Parallel Composition - LNT code*

The encoding presented in fig. 12 translates the *SubsetSelect* choreography construct into LNT process. Its behavior consists in making an inclusive choice between one or several outgoing branches to run at the same time. However a default path is specified in the case of no choice is made. In order to preserve the semantics of this construct, we have to be exhaustive and thus we take into account all the possible paths which are likely to occur during the execution of the process. Back to our running example, this means that we consider the possible execution of `proc_A` concurrently to the choice between `proc_B` or *null* - the terminating process, which does nothing - and reciprocally. Then, if no choice occurs, we execute the default path.

```
process Example_c [...] is
    select
        proc_A[...] || (proc_B[...] [] null)
        []
        proc_B[...] || (proc_A[...] [] null)
        []
        proc_Def[...]
    end select
end process
```

Fig. 12: *Subset Selection - LNT code*

Once the LNT processes and communication messages have been generated for each state in the choreography specification (itself expressed with the intermediate format), we compute the parallel composition of these processes. The resulting LNT-encoded choreography is then analysed using the CADP toolbox, and we perform some formal analysis so as to check interesting properties which are detailed in the following section.

V. Verification Package

This section presents some key-properties which need to be checked when designing choreography-based distributed systems. To ensure that these properties are satisfied, one can use model and equivalence checking techniques in a fully automated way. For instance, [1] already provides a verification tool suite for choreographies described via *BPMN 2.0* notation. We remind that our framework extends this previous work so as to support various choreography description languages as input. In practice, the key-properties we mainly focus on are respectively synchronizability, realizability, conformance and deadlock detection. The analysis of such properties in a first development step is crucial, because it may reveal design errors which could induce additional costs - *e.g.* changing a part of the implementation, debugging code... - if detected lately.

***Synchronizability**:* This is a property of a choreography ensuring that all peers can correctly synchronize with each others in both synchronous and asynchronous ways. Formally, a system is synchronizable if and only if the system behavior (over send actions) remains unaltered for any receive queue size [6]. Concretely, synchronizability checks that all interactions in the asynchronous system are also possible in the synchronous one.

This property is necessary for ensuring the realizability and conformance of possibly infinite systems (choreographies with loops). A recent decidability result [6] proposes a decision procedure for checking synchronizability. This result asserts that a system is synchronizable if and only if its behaviors assuming synchronous communication and asynchronous one, with interaction buffers bounded to size one, are equivalent.

Synchronizability is checked as follows. The choreography is first projected in order to generate the set of peers corresponding to the distributed implementation of the system. Then, on the one hand the system consisting of peers interacting synchronously is computed, and on the other hand the system consisting of peers interacting via 1-bounded FIFO buffers. Finally, equivalence checking is used for verifying that the two systems are equivalent. If this is the case the choreography is synchronizable.

***Realizability**:* This property checks that the distributed version of the system exactly behaves as specified in the choreography. This is crucial in a top-down development process in order to ensure that the implementation perfectly matches the global specification. Concretely, given the set of interacting peers obtained via projection (as previously done for synchronizability checking), it consists in verifying that the behaviour resulting from the parallel composition of all these peers matches the global choreography specification.

Realizability is checked as follows. First synchronizability must be verified to avoid analyzing a possibly infinite system. This may be the case if the choreography specification involves looping behaviours. Then, LTS models are computed from the choreography specification and from the set of interacting peers (the synchronous version is enough because our check

relies on synchronizability property). We finally compare these two LTSs and if they are equivalent (*i.e.* they generate the same behaviour), the choreography is realizable.

If the set of peers is not first synchronizable, one cannot decide whether the choreography is realizable for all interaction buffer sizes. However it is still possible to perform some bounded model checking, *i.e.* to set the buffer size to fixed value, and decide if the system is realizable for this specific case. Therefore the distributed implementation of a system can be realizable assuming synchronous communication whereas it is unrealizable assuming asynchronous one.

Conformance: This property is very close to realizability checking. The aim is to decide whether a set of peers generates the same behaviour (when running concurrently) as the one specified in the choreography. In realizability checking, we obtain the corresponding peers via projection according to the specification : this approach is rather integrated in a top-down design process.

At the opposite, in a bottom-up development process, peers are being reused and integrated into a new composition. The choreography serves as a contract that the implementation under construction must respect. From a verification point of view, it can be checked exactly as realizability, except that projection is not necessary. Conformance checking takes as input a choreography and a set of peers, whereas realizability checking only requires a choreography specification.

Deadlock detection: An other key-property that is relevant to verify is deadlock detection. A deadlock is commonly defined as a situation wherein two or more competing actions are each waiting for the other to finish, and thus neither ever does. Back to the choreography context, a deadlock is likely to occur when a peer locks its execution, ready to receive a message which the other peer will never send. The behaviour of the peers is represented using a LTS, and deadlocks are non final states with no outgoing transitions in the corresponding LTS. In such a configuration, the choreography designer has to revise the specification so as to avoid deadlock issues in the implementation.

VI. Tool Support

Basically, all of these properties can be checked using the CADP verification toolbox, thanks to an encoding of the intermediate format into the LNT process algebra, one of the CADP input languages. The verification of several properties (including the ones presented above) is fully automated through SVL scripts. These scripts are generated from the Python internal model and enable one to check precise properties according to the choreography LNT encoding - *e.g.* realizability, synchronizability...

Experiments: The following experiments have been achieved in [9] by *INRIA Convecs team* and give statistical information about the amount of data processed in CADP toolbox. These results show that the framework we implemented works for more than 200 choreographies. The experiments have been carried out on a Xeon W3550 (3.07GHz, 12GB RAM) running Linux.

For each experiment, table 13 gives first the specification language used for describing the input choreography and the size of the choreography : number of peers (P), interactions (Inter.), and selection operators (Sel.). Then, we give the size of the corresponding LTS and the size of the biggest intermediate state space for generating the asynchronous version of the distributed system (number of transitions and states). Finally, we provide the overall time for generating all LTSs (synchronous and asynchronous versions of the distributed system), and verifying synchronizability and realizability.

We have not checked conformance because we only used a choreography as input in our experiments (we remind that conformance checking requires a choreography and a given set of peers as inputs). The last column details the results for checking synchronizability (S) and realizability (R).

We can see that choreography specifications can result in huge LTSs (s̃everal thousands of states), mainly because parallel operators are expanded in all the possible interleaved behaviours when the corresponding LTS is generated. We also notice that the overall time for generating LTSs for choreography and both distributed systems (synchronous and asynchronous) as well as for verifying properties S and R is reasonable for medium-size choreographies. In other cases the analysis may take a lot of time due to the exhaustive exploration of all cases. The state space size raises quickly when the choreography implies several parallel behaviours.

Ex.	Lang.	|P|	|Inter.|	|Sel.|	$\|T\|/\|S\|$	Async. parallel compo. $\|T\|/\|S\|$	Time	Verif. S | R
1	Chor	3	10	1	21 / 29	127 / 200	48s	$\surd$ | $\surd$
2	BPMN	6	19	1	580 / 1,828	4,054 / 12,814	1m43s	$\surd$ | $\surd$
3	BPMN	6	19	1	18 / 20	750 / 3,298	1m40s	$\surd$ | $\surd$
4	BPMN	6	19	1	580 / 1,842	16,129 / 51,317	1m45s	$\surd$ | $\surd$
5	CP	7	11	1	11 / 11	158,741 / 853,559	5m47s	$\times$ | $\times$
6	BPMN	12	25	4	577 / 2,499	$\sim$1*10^6 / $\sim$7*10^6	8m43s	$\surd$ | $\surd$
7	BPMN	15	31	5	65,556 / 573,479	$\sim$2*10^6 / $\sim$18*10^6	1h34m	$\times$ | $\times$

Fig. 13: *Experimental Results [9]*

VII. Concluding Remarks

Team Contributions: In this work, we have implemented a framework for automating choreography verification. This analysis tool is divided into three parts. First, we have proposed an intermediate format used to describe choreographies in a formal way. Already existing interaction-based notations for such specifications (like BPMN 2.0 or Chor) have been connected to this intermediate format.

The intermediate format transposition to CADP toolbox is the second part of the framework. It consists in a Python library which generates primitives to translate choreography state machine into LOTOS NT process algebra, one of CADP input languages. This library was first implemented in [1] and some changes have been realised to support our input format specification. It now provides a connection between the intermediate format and our verification package.

The last module implemented is a verification package relying on the CADP toolbox. It presents a set of key-properties that choreographies must respect for ensuring realibility of the system under development. These properties are automatically tested in practice using model and equivalence checking techniques. This work, further detailed in [9], fosters the development of verification techniques and tools for formally analyzing choreographies.

Personal Contribution: We have mainly concentrated our efforts on the intermediate format development and also on the way it is connected to the Python library, while other *Inria Convecs team* members were providing the backend connection to the CADP toolbox.

Challenges: To this day, the BPMN 2.0 and Chor specification languages for choreographies are supported as inputs in our framework. As future works, connections to additional choreography description notations or to other verification tools could be achieved, so as to improve the modularity of our library. Currently it does not provide full support for the *DominatedChoice* construct, because of its encoding complexity.

An other interesting feature would be to extend our intermediate format to new constructs, in order to offer greater expressiveness to choreography modeling tools. This includes to consider data types and values in the specification. Such a feature requires to adapt the existing verification techniques to these kinds of variables.

Last, some features could be implemented in the checking package. If the choreography is not realizable, it would be useful to automatically integrate entities within the choreography specification. Such controllers would synchronize together in order to enforce the distributed system to respect the order of messages as specified in the global contract.

Acknowledgments

The author would like to thank Gwen Salaün (INRIA - Grenoble INP) and Matthias Güdemann (INRIA) for their work and their relevant comments, without whom this work would not have been possible.

References

[1] P. Poizat and G. Salaün, *Checking the Realizability of BPMN 2.0 Choreographies*. In Proc. of SAC'12, March 2012, Riva del Garda, Italy.

[2] G. Gössler and G. Salaün, *Realizability of Choreographies for Services Interacting Asynchronously*. In Proc. of FACS'11, September 2011, Olso, Norway.

[3] Q. Zongyan, Z. Xiangpeng, C. Chao and Y. Hongli *Towards the Theoretical Foundation of Choreography*. In Proc. of WWW'07, pages 973-982. ACM Press, 2007

[4] G. Salaün, L. Bordeaux and M. Schaerf, *Describing and Reasoning on Web Services using Process Algebra*. In Proc. of ICWS'04, IEEE Computer Society Press, pages 43-51, July 2004, San Diego, USA.

[5] G. Salaün and T. Bultan, *Realizability of Choreographies using Process Algebra Encodings*. In Proc. of IFM'09, LNCS 5423, Springer, pages 167-182, February 2009, Dsseldorf, Germany.

[6] S. Basu, T. Bultan and M. Ouederni, *Deciding Choreography Realizability*. In Proc. of POPL'12, pages 191-202, Philadelphia, Pennsylvania, USA, January 2012

[7] OMG, *Business Process Model and Notation – Version 2.0*. January 2011

[8] *PyXB Library Documentation*. http://pyxb.sourceforge.net. March 2012

[9] A. Dumont, M. Güdemann and G. Salaün, *VerChor : A Framework for Verifying Choreographies*. In Proc. of ICSOC'12, Submitted.