



A symbolic approach for the verification and the test of service choreographies

Hữu Nghĩa Nguyen Nguyễn

► To cite this version:

Hữu Nghĩa Nguyen Nguyễn. A symbolic approach for the verification and the test of service choreographies. Other [cs.OH]. Université Paris Sud - Paris XI, 2013. English. NNT : 2013PA112250 . tel-00961450

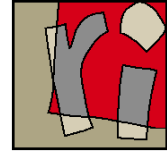
HAL Id: tel-00961450

<https://theses.hal.science/tel-00961450>

Submitted on 3 Apr 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



UNIVERSITÉ PARIS-SUD
ÉCOLE DOCTORALE INFORMATIQUE DE PARIS-SUD
LABORATOIRE DE RECHERCHE EN INFORMATIQUE

DOCTORAL THESIS

A Symbolic Approach for the Verification and the Test of Service Choreographies

PRESENTED ON OCTOBER 31, 2013

BY

Hữu Nghĩa NGUYỄN

TO RECEIVE THE DOCTORAL DIPLOMA OF COMPUTER SCIENCE

COMMITTEE IN CHARGE:

<i>Reviewers:</i>	Manuel NÚÑEZ	– Universidad Complutense de Madrid, Spain
	Gwen SALAÜN	– Grenoble Inp, Inria, France
<i>Examiners:</i>	Philippe DAGUE	– Université Paris Sud , France
	Pascal POIZAT	– Université Paris Ouest Nanterre la Défense, France
	Fatiha ZAÏDI	– Université Paris Sud, France
	Gianluigi ZAVATTARO	– Università di Bologna, Italy

SUPERVISORS: Pascal POIZAT and Fatiha ZAÏDI (dir.)

January 2010 – October 2013

Abstract

Service-oriented engineering is an emerging software development paradigm for distributed collaborative applications. Such an application is made up of several entities abstracted as services, each of them being for example a Web application, a Web service, or even a human. The services can be developed independently and are composed to achieve common requirements through interactions among them. Service choreographies define such requirements from a global perspective, based on interactions among a set of participants. This thesis aims to formalize the problems and attempts to develop a framework by which service choreographies can be developed correctly for both top-down and bottom-up approaches. It consists in analyzing the relation between a choreography specification and a choreography implementation at both model level and real implementation level. Particularly, it concerns the composition/decomposition service design, the verification, and the testing of choreography implementation. The first key point of our framework is to support value-passing among services by using symbolic technique and SMT solver. It overcomes false negatives or state space explosion issues due by abstracting or limiting the data domain of value-passing in existing approaches. The second key point is the black-box passive testing of choreography implementation. It does not require neither to access to source codes nor to make the implementation unavailable during the testing process. Our framework is fully implemented in our toolchains which can be downloaded or used online at address: <http://schora.lri.fr>.

Keywords: Choreography, Web services, Value-passing, Conformance, Projection, Realizability, Passive testing, Symbolic transition systems, Tools.

Résumé

L'ingénierie orientée services est un nouveau paradigme pour développer des logiciels distribués et collaboratifs. Un tel logiciel se compose de plusieurs entités, appelés services, chacun d'entre eux étant par exemple une application Web, un service Web, ou même un humain. Les services peuvent être développés indépendamment et sont composés pour atteindre quelques exigences. Les chorégraphies de service définissent ces exigences avec une perspective globale, basée sur les interactions entre des participants qui sont implémentés en tant que services. Cette thèse vise à formaliser des problèmes et tente d'élaborer un environnement intégré avec lequel les chorégraphies de services peuvent être développés correctement pour les deux types d'approches de développement : l'approche descendante et l'approche ascendante. Elle consiste à analyser la relation entre une spécification de chorégraphie et une implémentation de la chorégraphie au niveau du modèle et aussi au niveau de l'implémentation réelle. Particulièrement, il s'agit de la composition/décomposition des services, la vérification, et le test de l'implémentation de chorégraphie. Le premier point-clé de notre environnement intégré est de représenter le passage de valeurs entre les services en utilisant la technique symbolique et un solveur SMT. Cette technique nous permet de réduire les faux négatifs et de contourner le problème d'explosion combinatoire de l'espace d'états, ces problèmes sont durs à l'abstraction et à l'énumération des valeurs pour les approches existantes basées données. Le second point-clé est le test passif boîte noire de l'implémentation de chorégraphie. Il ne nécessite pas d'accéder au code source, ni de rendre indisponible l'implémentation pendant le processus de test. Notre environnement intégré est mis en œuvre dans nos outils qui sont disponibles en téléchargement ou à utiliser en ligne à l'adresse : <http://schora.lri.fr>.

Mots-clés : Chorégraphie de services, Service web, Passage de valeurs, Conformité, Projection, Réalisabilité, Test passif, Système de transitions symboliques, Outils.

Acknowledgment

I would not have been able to complete this dissertation without the help and support of the kind people around me, to only some of whom it is possible to give particular mention here.

First and foremost, I would like to present my deepest gratitude to my supervisors: Prof. Pascal POIZAT, and Dr. Fatiha ZAÏDI, directress of my thesis, for their excellent guidance, patience, support, and encouragement.

Beside my supervisors, I would like to thank the rest of my thesis committee: Prof. Philippe DAGUE, Prof. Manuel NÚÑEZ, Dr. Gwen SALAÜN, and Prof. Gianluigi ZAVATTARO, for taking their time to review carefully my thesis and for their insightful comments.

Special thanks are dedicated to Prof. Philippe DAGUE, director of LRI, for his welcome and support during my PhD study.

I thank my labmates in LRI for the stimulating discussions.

Last but not he least, I thank my family, parents, brother and sisters. They are always supporting me and encouraging me with the best wished.

Contents

	Page
Contents	iv
List of Figures	vii
List of Tables	ix
List of Listings	xii
List of Acronyms	xiii
1 Introduction	1
1.1 General Context	2
1.2 Problem Statement	3
1.3 Overview of Our Approach	4
1.4 Contributions of the Thesis	5
1.5 Publications	6
1.6 Outline of the Thesis	7
2 Service Choreographies: Issues & Challenges	9
2.1 Service Choreography	10
2.1.1 Coordination Languages and Models	10
2.1.2 Service Choreography	12
2.1.3 Choreography Development: Issues and Challenges	13
2.2 Choreography Modeling	15
2.2.1 Basic Elements	16
2.2.2 Data Support	19
2.2.3 Communication Modes	20
2.2.4 Parallelism	21
2.3 Realizability & Projection	21
2.3.1 Realizability Notions	21
2.3.2 Realizability through Projection	23

2.3.3	Causes of Unrealizability	23
2.3.4	Enforcing Realizable Choreographies	24
2.3.5	Impact of Data on Realizability	25
2.4	Conformance Checking	26
2.4.1	Choreography Conformance	26
2.4.2	Conformance Relation	27
2.4.3	Impact of Data on Conformance Relation	28
2.5	Choreography Testing	29
2.5.1	Formal Software Testing	29
2.5.2	Service Choreography Testing	31
3	A Symbolic Model of Choreographies	35
3.1	Specification Language	36
3.1.1	Basic Events: Interaction <i>vs.</i> Sending & Reception	36
3.1.2	General Language	37
3.2	Symbolic Semantics	39
3.2.1	Symbolic Transition System	39
3.2.2	Transformation Rules	40
3.2.3	STG Product	41
3.2.4	Reachability	43
3.3	Symbolic Conformance	43
3.3.1	Making Implementation and Specification Comparable	44
3.3.2	Conformance Relation	44
3.3.3	Conformance Computation	47
3.3.4	PES Satisfiability and Conformance Verdict	48
3.4	Realizability Checking & Projection	51
3.4.1	Event connectedness	54
3.4.2	Data connectedness	55
3.4.3	Branching decision	56
3.5	Tool & Experimental Evaluations	58
3.5.1	Boolean Condition Solver	59
3.5.2	Tool Architecture	59
3.5.3	Experimental Evaluation	60
3.6	Discussion	65
4	Passive Testing of Choreographies	67
4.1	Passive Conformance Testing	68
4.1.1	<i>Chor</i> Language & Trace Semantics	68
4.1.2	Local & Global Conformance	70
4.1.3	Implementation	72
4.1.4	Observation of SOAP Messages	74

4.1.5	Global Log Synthesis	75
4.1.6	Testing Algorithm	76
4.1.7	Tool & Experimental Evaluation	77
4.2	Online Property-Oriented Testing	81
4.2.1	Symbolic Transition Graph with Assignments	82
4.2.2	Local Properties	86
4.2.3	Global Property	89
4.2.4	Implementation	90
4.2.5	Experimental Evaluation	96
4.3	Discussion	97
5	Case Study	99
5.1	Case Study Description	100
5.2	Verification	100
5.2.1	Reachability Checking	100
5.2.2	Realizability Checking	102
5.2.3	Choreography Projection	102
5.2.4	Conformance Checking	103
5.3	Testing	105
5.3.1	Conformance Testing	106
5.3.2	Property-Oriented Testing	108
6	Conclusions and Perspectives	111
6.1	Synthesis of Results	112
6.2	Perspectives	114
A	SChorA Syntax	117
A.1	Declaration Part	117
A.2	Command Part	118
	Bibliography	121

List of Figures

2.1	Coordination of Services	10
2.2	Coordination Modeling Approaches of Online Shopping Process	12
2.3	Choreography, Roles, Services and Global, Local Viewpoints	13
2.4	Issues in Choreography Development Process	14
2.5	Examples of unrealizable choreographies	14
2.6	Criteria of Choreography Modeling	16
2.7	Example of Interconnection and Interaction-based Modeling Approaches	17
2.8	Impact of Communication Modes: Realizability in Synchronization \rightarrow Unrealizability in (a) Asynchronous–Sender, (b) Asynchronous–Receiver, and (c) Asynchronous–Disjoint	24
2.9	Impact of Data: unrealizability \rightarrow realizability	26
2.10	Conformance Checking Scheme	27
2.11	Impact of Data on Conformance Relation: Maybe Verdict	29
2.12	Active and Passive Testing Processes	31
2.13	An Architecture of Distributed Testing	33
3.1	Semantics of STG	40
3.2	Transformation from our Language to STGs	40
3.3	Rules for the Product of STGs	41
3.4	STGs for Example 1 and Example 2	42
3.5	Two Refinements for the Implementation in Figure 3.4(d)	45
3.6	Restrictions of the STGs in Figure 3.5	45
3.7	Retrieval of $\xRightarrow{\tau}$ Transitions and Their Semantics	46
3.8	Online Shopping Process in (a) Extending BPMN 2.0 Choreography [Knup- plesch et al., 2012] and in (b) Symbolic Transition Graph	52
3.9	The Projection of STG in Figure 3.8(b) on Role: (a) buyer, (b) vendor, (c) warehouse, and (d) shipper)	53
3.10	Projection with Additional Interactions of STG in Figure 3.8(b) under Synchronization Communication Mode	58
3.11	Architecture of our Toolchain for Conformance Checking	60
3.12	Architecture of our Toolchain for Choreography Projection	61

3.13	Impact of Number of (a) States, (b) Branches, (c) Roles, (d) Operations and (e) Variables on the Realizability of Choreography; and (f) Verification Time	63
4.1	<i>Chor</i> Choreography Language	68
4.2	Operators on Traces	69
4.3	<i>Role</i> language	70
4.4	Natural Projection of <i>Chor</i> Language	70
4.5	Conformance Testing Choreography Implementations	73
4.6	Sendings and Receptions Correlation	76
4.7	Tester Scalability	80
4.8	STGAs of four services	84
4.9	Shipping choreography	85
4.10	Trace Semantics of STGA	85
4.11	Architecture of the Verification System	91
4.12	Online Verification of Local Property	92
4.13	Tester Scalability	96
5.1	Reachability Checking by SChorA tool	101
5.2	Realizability Checking by SChorA tool	102
5.3	Choreography Projection by SChorA tool	103
5.4	Conformance Checking by SChorA tool	104
5.5	Conformance Checking of a Mutation of the Case study	105
5.6	Point of Observation of SOAP-Capturer (1) and SOAP-Forwarder (2) . .	106
5.7	Graphic User Interface of SOAP-Forwarder	106
5.8	Conformance Testing of an Implementation	108
5.9	Verdict Output of Vendor Service Testing	109

List of Tables

2.1	Interaction <i>vs.</i> Interconnection in choreography languages	18
2.2	Interaction <i>vs.</i> Interconnection in Choreography Models	18
2.3	Data Support in Choreography Modeling Approaches	20
2.4	Service Choreography Projection Approaches	23
2.5	Choreography Conformance Relations	28
2.6	Service Choreography Testing Approaches	32
3.1	The Basic Events	37
3.2	Decision Table for Conformance based on PES Satisfiability Checking . .	50
3.3	Natural Projection of STG of Choreography	53
3.4	Experimental Results of Conformance Checking	61
3.5	Experimental Results of Projection & Realizability Checking	62
4.1	Online-Shopping Case Study	78
4.2	Semantics of Global Property	90

List of Algorithms

1	Product of n STGs	42
2	Reachable STG	43
3	Conformance PES Computation	49
4	Correction of Data Connectedness	56
5	Correction of Branching	57
6	Synthesis of Global Observations (<i>synthesisObservations(L)</i>)	77
7	Preorder Verification (<i>isPreorder(l, t)</i>)	78
8	Global Conformance Verification	78
9	Checking Correctness of Local Property	93
10	Translation of Local Property to XQuery	95

List of Listings

3.1	Translation into the Z3 Language of the PES in Example 5	50
3.2	Example of Condition Solver using Z3 SMT	59
4.1	Example of Complex Message Exchange in XML	82
4.2	Example of Captured Message	94
4.3	Example of Transformation of Local Property into XQuery	95
5.1	Example of Captured log of Vendor Service	107
5.2	A Local Property to Verify Vendor Service	108
A.1	Structure of SChorA Script	117
A.2	Syntax of Model Descriptions Used in SChorA Script	118
A.3	Syntax of Commands Used in SChorA Script	118
A.4	Example of SChorA Script	120

List of Acronyms

SOAP	Simple Object Access Protocol.....	3
IUT	Implementation Under Test.....	4
PO	Point of Observation.....	4
STG	Symbolic Transition Graph.....	5
WS-BPEL	Web Service Business Process Execution Language.....	11
WS-CDL	Web Service Choreography Description Language.....	11
BPMN	Business Process Model & Notation.....	14
MSC	Message Sequence Chart.....	16
BPEL4Chor	Choreography Extension for BPEL.....	16
CD	Collaboration Diagram.....	16
UML	Unified Modeling Language.....	16
CSP	Communicating Sequential Processes.....	18
FIFO	First In, First Out.....	21
PCO	Point of Control and Observation.....	30
STGA	Symbolic Transition Graph with Assignments.....	81
CE	Candidate Event.....	87
SChorA	Symbolic Choreography Analysis tool.....	99
OSP	Online Shopping Process.....	100
SBBC	Symbolic Branching Bisimulation for Conformance.....	103
PES	Predicate Equation System.....	105
Prop-tester	Property-Oriented Testing tool.....	105
PACT	Passive Conformance Testing tool.....	106
STS	Symbolic Transition System.....	115

Introduction

Contents

1.1	General Context	2
1.2	Problem Statement	3
1.3	Overview of Our Approach	4
1.4	Contributions of the Thesis	5
1.5	Publications	6
1.6	Outline of the Thesis	7

1.1 General Context

Service-oriented engineering is an emerging paradigm for the development of distributed applications. This trend has increased with the emergence of component and service architectures such as Web services. Such an application is made up of several entities abstracted as *services*, each of them being for example a Web application, a Web service, or even a human. To reach a common objective, the services have to coordinate by interacting with each other. A centralized point of view may be taken on collaboration, which nicely suits service orchestration. However, modern processes and applications are much more collaborative in nature. Hence, they should be specified and implemented in a collaborative way too. This is where service choreography may help.

Choreographies support a collaborative vision of a distributed application. It specifies from a global perspective the interactions between *roles* played by services in some collaboration. A choreography is a specification of what the collaboration participants, or roles, should follow and achieve altogether. Due to its global perspective, a choreography focuses on *interactions* between two roles.

Besides the global perspective, the interactions may be seen from the local perspective of each role. From this perspective, only interactions that directly involve the role are captured. Consequently, there exist two kinds of models, *local models*, or *role models*, which specify the local behaviors of roles (one for each), and *global models*, or *choreography models*, that correspond to choreography specifications.

Choreography models are useful during the early phases of system analysis and design thanks to their global perspective, while role models are blueprints for the implementation of services realizing roles, for the derivation of test cases for these services, and for the adaptation of reused services to the choreography constraints. Consequently *bridging the gap between choreography and role models* is a cornerstone for top-down choreography development processes. This relates to the *projection* issue which generates relevant role models, from a choreography model. There exist two type of projection, *natural projection* which generates local models from local one without introducing any additional interactions whereas *smart projection* may introduce some additional interactions among generated local models, *e.g.*, when the choreography is not realizable. In other words, the natural projection is correct when the choreography is realizable. The *realizability* attribute of a choreography is whether the choreography model can be correctly projected to role models, *i.e.*, the composition of generated models conforms to the choreography.

Another issue of choreography is the relation with implementation. Services may be written from scratch or be reused, and are then coordinated to fulfill the choreography.

This relates to automatic service composition and orchestration. An alternative is the generation of service skeletons from choreographies, *e.g.*, thank to the projection. Skeletons are then completed by developers to build a running system. In any case, an important issue, so-called *conformance*, is to check whether the implementation exhibits or not the behaviors specified in the choreography. When service code is available, or when behavioral interfaces of the services are provided, verification using model-checking or behavioral equivalences is an alternative. In the opposite case, formal testing is of great help. It enables one to check whether an implementation conforms or not to a specification without requiring an access to its code, *e.g.*, the implementation source code is unavailable.

1.2 Problem Statement

Lack of Value-Passing Support in Choreography Models. When model of choreography and its implementation are available, some issues of choreography development may be done at model level. Choreography interactions are usually achieved through message exchanges, *e.g.*, Simple Object Access Protocol (SOAP) messages in Web services. It is critical to support value-passing in modeling and then in analyzing service choreographies.

Most existing approaches do not adequately support the value-passing, *i.e.*, without explicitly considering the data exchanged through interactions and using it for branching decisions. They just *abstract away from data*. This may yield *over-approximation issues*, *e.g.*, false negatives in verification process. Hence the presence of value-passing in choreography model may change its attributes, *e.g.*, realizability.

Some approaches do this by working on *closed implementation-level systems*, *i.e.*, value-passing is only with ground data. In such a case, this could lead into serious efficiency problems, *state space explosion*, when analyzing choreography. Some others avoid this problem by analyzing choreography based on the syntax when checking realizability. However, these approaches miss the cases of unreachable and conditional. Moreover, the models used in these works are interaction-based but data are expressed as the way of interconnection-based, *i.e.*, the variables are defined at each local role then their values are synchronized thanks to interactions. This is not adequate for an interaction-based approach. The interactions together with variables should be the basic events.

Limitation of Control and Observation of Choreography Implementation.

After the system has been implemented, the implementation must be verified to conform to its specification, to ensure that the system will operate correctly. When the model of implementation is not available, the conformance between the imple-

mentation and its choreography specification may be guaranteed by testing, also known as *conformance testing*.

Testing consists in performing experiments on Implementation Under Test (IUT). An implementation of choreography is a set of services which can be independently developed, then composed to realize a common goal. The implementation is usually deployed in a distributed system composed of loosely coupled machines which are physically distributed and do not share system resources but rather are connected via message exchanges.

In distributed context, *it is difficult, even impossible in some situation, to control overall IUT, e.g., system working 24/7* where the IUT can be running in their real environment. Therefore, the testing should not disturb its natural operation, since it might produce a wrong behavior of the overall system. In other words, the tester should realize the testing only by observing the behavior of IUT, *e.g., inputs and output IUT, through Point of Observations (POs)*.

The observations might be not complete. It is caused by unobservable events or partial order of occurrences of observations, *i.e., lack of shared clock of distributed system*. Consequently, the tester only has a partial view or a set of partial views of the IUT.

1.3 Overview of Our Approach

Symbolic Technique in Modeling and Verifying Service Choreographies.

In our framework, data is supported using a *symbolic approach and the use of an SMT solver*. Since the model is equipped with data and loops, we use symbolic techniques in order to avoid the usual state space explosion problem (when messages parameters or variables are flattened *wrt.* their infinite domains). Hence, messages parameters and variables are represented by symbolic values instead of concrete ones. Symbolic models and equivalences enable us to analyze choreographies in presence of data without suffering from state space explosion and without bounding data domains.

Passive Testing to Test Choreography Conformance. Passive testing is a testing approach which aims to not disturb execution of IUT during testing process. Contrarily, active testing requires to control IUT by sending some inputs to IUT and by observing its outputs. Passive testing detects faults “in-process”, *i.e., while the IUT is in its normal operation*. The exchanged messages between services of IUT will be recorded in a log that will be examined later on against the properties derived from experts.

1.4 Contributions of the Thesis

Our contributions are manifold. They can be grouped into three major categories. The first contribution is *a symbolic model and framework to specify and analyze service choreographies*. Particularly, we define a formal language with an interaction model addressing both global (choreography) and local (role requirements, service description) perspective. Our language supports *information exchange and data-related constructs* (conditional and loop constructs). We give also a *fully symbolic semantics* to this language using a model transformation into Symbolic Transition Graphs (STGs), thus avoiding data abstraction and over-approximation, restriction by manually bound data domains, and limitation to implementation-level closed descriptions.

Based on the developed model, we then propose a symbolic framework in which the data variable are manipulated by using symbols rather than their concrete values. This enables one to model and analyze service choreography in presence of data without suffering from state space explosion and without bounding data domains. The framework is used to deal with realizability checking, conformance checking, and projection.

We do not only check whether the choreography is realizable, but also we propose solutions for rendering it realizable. For that purpose, we build a *smart projection* function, dealing with data. It is dedicated to enable the realizability of choreography, *i.e.*, when the choreography is unrealizable, some extra interactions can be automatically introduced. Furthermore, the minimum extra interactions are added in order to obtain minimal implementation and traffic. *The projection considers both synchronous and asynchronous communication modes.*

We build on branching bisimulation and on a symbolic extension of weak bisimulation to develop a specific *symbolic version of branching bisimulation* dedicated at *checking the conformance* of a set of local entities *wrt.* a choreography specification. Our equivalence enables one to check conformance in presence of choreography *refinement*, *i.e.*, where new services and/or interactions may be added *wrt.* the specification. Going further than a *true* vs. *false* result for conformance, our approach supports the generation of the *most general constraint over exchanged information* in order to have conformance.

The second contribution is related to passive testing of service choreographies. It tackles the peculiarities of choreography implementation through non-intrusiveness, support for *black-box services without source code being available*. Our work is started by passive conformance testing of service choreography. Several languages have been proposed for choreography. We chose *Chor* since it is both expressive and abstract

enough to suit the requirements of a specification language. This work is based on naïve passive testing approach, *i.e.*, any execution traces of implementation will be examined against choreography specification in order to emit verdicts. *Conformance is checked both at the local and at the global level.* Local conformance represents whether some service implementation plays or not correctly its role in choreography. Global conformance ensures that interactions between services follow the prescription of the choreography or if they diverge from the envisioned collaboration.

The limitations of the testing approach above, *e.g.*, offline testing, without value-passing, and requirement of global clock to synthesize, have been overcome by the second one. It is based on property-oriented passive testing approach, *i.e.*, only some execution traces which concern tested properties are examined. *A property can express a critical, positive or negative, behavior to be tested on an isolated server (locally) or on a set of services (globally).* This work supports *online verification*, *i.e.*, faults are detected as soon as they are generated, of these kind of properties are checked against local running traces of each service in a distributed system where *no global clock is needed.*

The last contribution of the thesis is *the availability of toolchains.* They demonstrate our theoretical approaches, *e.g.*, symbolic choreography analysis–SChorA¹, and choreography testing². The toolchains and their source code are totally available for using and developing. Furthermore, SChorA has a Web application version thus one can use it immediately anywhere by using any web browsers without installation and configuration.

1.5 Publications

Main contributions of this thesis have already been published in proceedings of international conferences as well as presented in national research days.

International Conferences.

1. Huu Nghia Nguyen, Pascal Poizat and Fatiha Zaïdi. *Automatic Skeleton Generation for Data-Aware Service Choreographies.* in ISSRE - IEEE International Symposium on Software Reliability Engineering, pages 320-329. November 2013.
2. Huu Nghia Nguyen, Pascal Poizat and Fatiha Zaïdi. *Online Verification of Value-Passing Choreographies through Property-Oriented Passive Testing.* in HASE -

¹<http://schora.lri.fr>

²<https://www.lri.fr/~nhnghia/tools>

IEEE International Symposium on High Assurance Systems Engineering, pages 106-113. October 2012.

3. Huu Nghia Nguyen, Pascal Poizat and Fatiha Zaïdi. *A Symbolic Framework for the Conformance Checking of Value-Passing Choreographies*. in ICSSOC - International Conference on Service Oriented Computing, pages 525-532. November 2012.
4. Huu Nghia Nguyen, Pascal Poizat and Fatiha Zaïdi. *Passive Conformance Testing of Service Choreographies*. in SAC - ACM Symposium on Applied Computing, pages 1528-1535. March 2012.

Talks.

1. *A Symbolic Framework for the Conformance Checking of Value-Passing Choreographies*. in Journées du GDR GPL, MTV2 track. April 2013.
2. *Vérification des chorégraphie implantant en BPEL*. in Journées du GDR GPL, COSMAL track. June 2011.

Poster.

1. *Symbolic Approach for the Verification and the Test of Service Choreographies*. in Journées du GDR GPL. April 2013.

1.6 Outline of the Thesis

The rest of the thesis is divided into five chapters. Basic notions of service choreographies and related work are presented in Chapter 2 while the main contributions of the thesis are presented in Chapter 3 and Chapter 4. Particularly, they are organized as follows:

Chapter 2 presents a state of the art of service choreographies. We go from the general notions of service choreographies to an overview of choreography modeling approaches and choreography testing approaches.

Chapter 3 introduces our formal language and model for interaction-based service choreographies with value-passing. The symbolic semantics of the

language and model are also introduced by translating them into Symbolic Transition Graph. Based on the proposed model, we examine the fundamental issues of choreography, *e.g.*, conformance, projection and realizability with the presence of value-passing in the model. Finally, we introduce our tools to validate the proposed model.

- Chapter 4 presents two passive testing approaches: one naïve approach and another one property-oriented approach, to test real implementations of service choreographies. The naïve approach tests all observed behaviors of implementation while the property-oriented approach focuses only on some critical behaviors of the implementation. There are also some improvements from the former to the latter, *e.g.*, no data to data, offline to online, global clock to no global clock. The evaluations of our work by tool supported are also given.
- Chapter 5 presents an application of our toolchain, which is presented in the chapters above, on development process of a simple case study. This consists of specification and modeling the case study by our language and model. It is then verified by SChorA (Symbolic Choreography Analysis) tool which analyzes, by using symbolic approach, some attributes such as the realizability, the reachability, the conformance, and the projection. Finally, two passive testing tools, one online and one offline, will be introduced to guarantee the correctness of implementation.
- Chapter 6 concludes this thesis. The limitations of the thesis are discussed and some future works are also pointed out.

Service Choreographies: Issues & Challenges

Contents

2.1	Service Choreography	10
2.2	Choreography Modeling	15
2.3	Realizability & Projection	21
2.4	Conformance Checking	26
2.5	Choreography Testing	29

The object being studied of the thesis is service choreographies. This section is dedicated to present their definitions as well as the fundamental issues in choreography development process, *e.g.*, realizability, projection, conformance. We then present, compare and point out limitations in existed researches which try to solve one or several of these issues.

2.1 Service Choreography

2.1.1 Coordination Languages and Models

Nowadays, a software system is more and more complex. It usually consists of many software components, called *services*. The computation of the system is thus divided in each its services. The services are usually implemented in distributed environment. They communicate with each others to achieve common requirements.

To build such a system, there exist two approaches, top-down and bottom-up. The *top-down approach* bases on the decomposition of a system into services which can be put into practice. Contrarily, the *bottom-up approach* bases on composition of existing services to achieve requirements of the system.

Coordination languages and models are developed to deal with the problem of managing interactions among services, which are concurrent and distributed processes, hence some drawbacks can occur: deadlocks, starvation or incorrect multiple access to resources, ... So that, the coordination is the consistent organization of the communication and its effects, such that required cooperation between all services involved is established.

To model such a system, two parts are taken into account: computation and coordination. The computation model helps programmers only to build a single computational service. Whereas the coordination model is the glue that binds separate services into an ensemble.



Figure 2.1: Coordination of Services

Exogenous vs. Endogenous. The coordination might be integrated into a computational service or be implemented by an entity separating with the computational service. The former is called endogenous coordination, while the latter is called exogenous coordination. Consequently, we have endogenous and exogenous coordination languages and models which are defined in [Arbab, 1998] as following:

“*Endogenous* coordination languages and models provide coordination primitives that must be incorporated *within* a computation for its coordination. In applications that use such models, primitives that affect the coordination of each module are inside the module itself.”

“*Exogenous* models and languages provide primitives that support the coordination of entities from *without*. In applications that use exogenous models primitives that affect the coordination of each module are outside the module itself.”

For instance, Linda is an endogenous model, whereas Reo [Arbab, 2004] is an exogenous one. [Capizzi et al., 2004] shows that a distributed system can be moved from an underlying endogenous data-driven to an endogenous event-driven coordination model, simply by replacing the coordination aspects and leaving the computation code unchanged.

Orchestration vs. Choreography. Depending on the interaction topology in coordination model, service compositions are classified into two styles: orchestrations and choreographies [Peltz, 2003]. *Orchestration* always represents control from one participant’s perspective, called *orchestrator*. It differs from choreography which is more collaborative and addresses the interactions that implement the collaboration among participants [Meng and Arbab, 2007]. Unlike the orchestration, there is no privileged entities in the choreography – which is a peer-to-peer set of relationships, without looking at any single participant’s internals. The coordination of a set of services in an orchestration system can be considered as exogenous coordination, *i.e.*, thanks to orchestrator. Meanwhile, in choreography, each participant controls itself in the collaboration in regarding activities of others.

Figure 2.2 illustrates topologies of interactions of orchestration and choreography approaches to model an Online Shopping Process which has three participants: buyer, vendor and shipper. The choice between an orchestration and a choreography approach to model this collaboration may be driven by a number of factors, often of an organizational nature. For instance, to easily implement and closely monitor the performance of the participants, it is desirable to have a single point, orchestrator in Figure 2.2(a), of interaction between them. This orchestrator would handle all interactions related to the collaboration. Thus, interactions between the buyer, the vendor and the shipper become the ones between buyer and the orchestrator and the ones between the orchestrator and the vendor and the shipper. In orchestration approach, the collaboration depends on the orchestrator which may introduce some risks, *e.g.*, overload, resource location, maintenance, ... The drawbacks may be overcome in choreography approach. However the choreography approach faces the difficulty of management and monitoring the participants due to their distributed environment.

An orchestration can be transformed into a choreography and vice versa. [McIlvenna et al., 2009] investigates about the synthesis of orchestrators from choreography. A translation from Web Service Choreography Description Language (WS-CDL) to Web

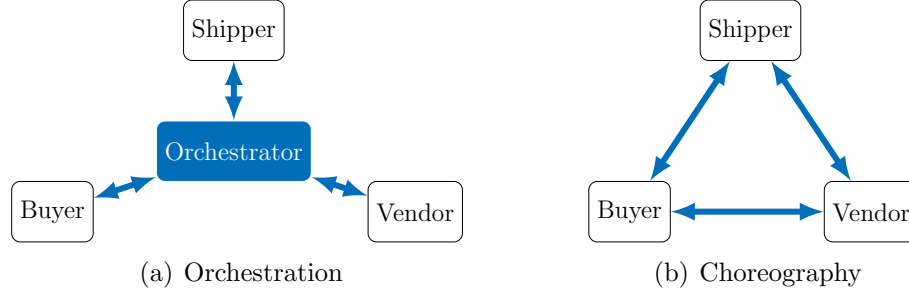


Figure 2.2: Coordination Modeling Approaches of Online Shopping Process

Service Business Process Execution Language (WS-BPEL), which is a programming language to build orchestrators, is studied by [Mendling and Hafner, 2008]. [Quinton et al., 2009] proposes to build, whenever possible, orchestrators, and then distribute them using protocols so as to obtain choreography.

2.1.2 Service Choreography

Choreography, Role and Service. *Choreography* is the specification, from global perspective, of interactions between participants in some collaboration. A choreography is a constraint that each service implementing a role has to follow.

An *interaction* represents a communication between a participant, *sender*, to other participants, *receiver(s)*, in a choreography. An interaction represents usually a message exchange. Most choreography modeling approaches consider only the interactions between two participants, *i.e.*, there exists only one receiver. Thus an interaction from sender to a set of receivers can be represented by a set of interactions, in parallel, from the sender to each receiver.

“A *Role* represents the behavior that a *participant* has to exhibit in order to fulfill the activity defined by the choreography” [Busi et al., 2006]. A role can be implemented by one or several *service(s)*, which can be also called *peers*, which represents an active component which may be identified with a process, an active object, a thread, a Web service, a user, an organization, ... A service can be implemented by one or several roles. In this thesis, we considered the case 1 role – 1 service, *i.e.*, a role is implemented by one and only one service. In our approach, role, service and interaction are atomic elements.

Global & Local Models. Besides the global perspective of choreography, interactions may be seen from the *local perspective* of each role. From this perspective, only interactions, that directly involve the role, are captured. Consequently, there exist

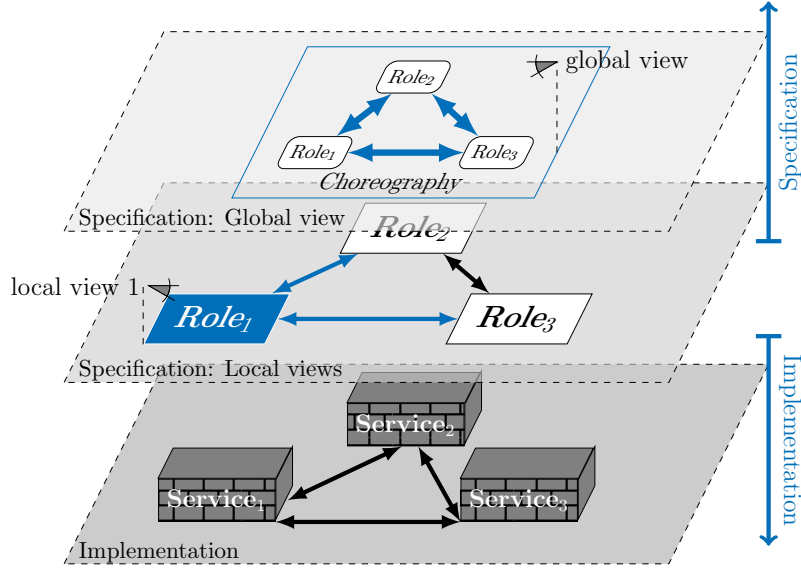


Figure 2.3: Choreography, Roles, Services and Global, Local Viewpoints

two kinds of models, *local models*, or *role models*, which specify the local behaviors of roles (one for each), and *global models*, or *choreography models*, that correspond to choreography specifications. Choreography models are useful during the early phases of system analysis and design thanks to its global perspective, while role models are blueprints for the implementation of services realizing roles, for the derivation of test cases for these services, and for the adaptation of reused services to the choreography constraints.

The notion of local model is identical with the one of “orchestrator” in [Busi et al., 2006, Li et al., 2007b]. Indeed, regarding Figure 2.2(b), if we consider a collaboration which captures only the interactions between the buyer and the vendor, the shipper, *i.e.*, it does not capture the interactions between the vendor and the shipper, then the buyer is an orchestrator in this collaboration. In the other hand, this collaboration represents the local view point of the buyer in the choreography. Hence the interactions in this collaboration is also represented by local model of the buyer.

2.1.3 Choreography Development: Issues and Challenges

A choreography is a description of a coordination system which can be implemented based on top-down or bottom-up approaches. Figure 2.4 illustrates the development process of service choreography. In this section, we give a brief of the three fundamental issues in choreography development process. Their details will be introduced

in the next sections. Some challenges to solve them will be also pointed out.

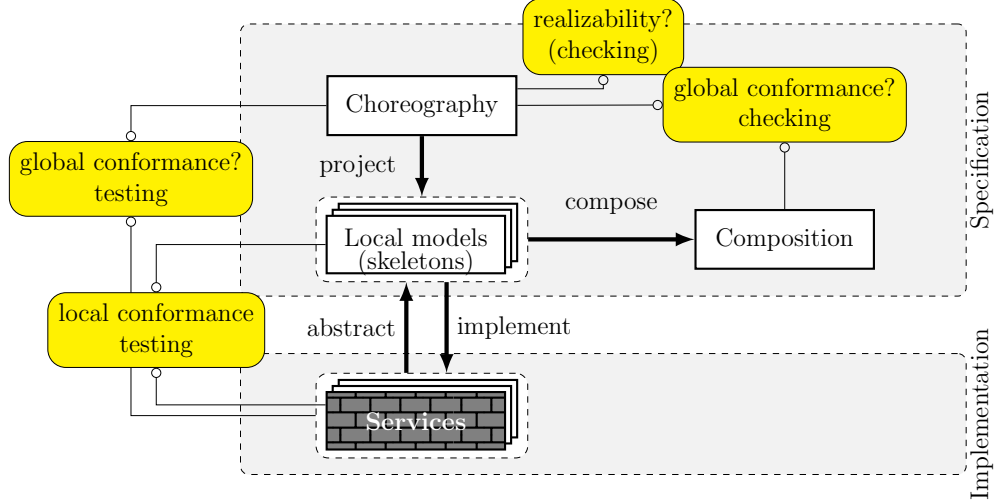


Figure 2.4: Issues in Choreography Development Process

Issues. The first issue is *realizability* which intends to check *whether a choreography can be implemented?*. Generally, not all choreographies are realizable. Let us describe, by using Business Process Model & Notation (BPMN) 2.0 notations, two unrealizable choreographies as in Figure 2.5. The left hand side choreography describes a **request** interaction between two roles *buyer* and *vendor* followed by a **ship** interaction between roles *warehouse* and *shipper*. This choreography is not realizable since role *warehouse* has no possibility to know when **ship** must be done. It is done after **request**, but *warehouse* never knows when **request** occurs. The right hand side choreography requires that either **request** or **ship** is done but there is no way to ensure that since different senders, *i.e.*, *buyer* and *warehouse*, are concerned. The approaches for realizability checking will be presented in Section 2.3.

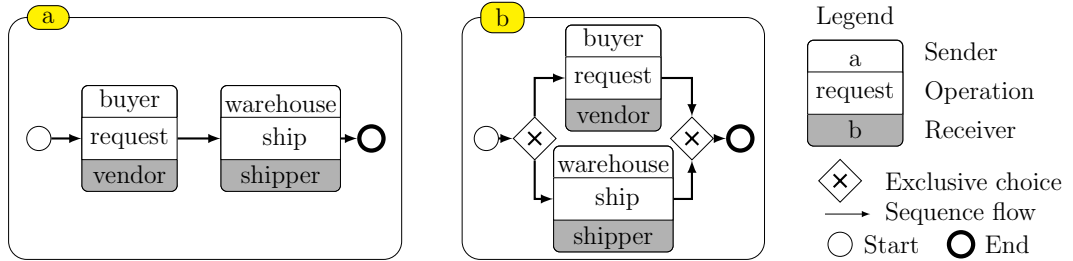


Figure 2.5: Examples of unrealizable choreographies

The second issue is projection. In top-down approaches, if the choreography is realizable, it is then projected on each role to obtain role model from which each service is implemented. *Projection* can be defined as a procedure which takes as an input a choreography model with n roles and outputs a set of n local models.

To ensure the correctness of projection, the conformance between the choreography model and the composition of the generated role models, called *global conformance* should be verified. The third issue is *conformance*. The global conformance checking is the cornerstone also in bottom-up approaches. In this approach, the participant services in the implementation of the choreography are selected from existed services thanks to its model. To guarantee the collaboration of selected services with respect to the choreography, the global conformance should be also checked.

When model of participant services are not available, conformance testing can be done to verify the correctness of implementation. *Local conformance testing* ensures a service plays correctly its roles while *global conformance testing* guarantees the collaboration between services with respect to the choreography.

Challenges. The issues above can be solved at model level, *e.g.*, projection, conformance checking between role models and choreography model, or at implementation level, *e.g.*, conformance testing between set of services and the choreography when models of participant services are not available,.

The first challenge is a *lack of data support* in existing choreography modeling approaches. Since the interactions are usually achieved thanks to message exchange, *value-passing and data should be supported* in choreography models, and in the solving processes of the issues above. Some approaches just *abstract away* from data. This may yield over-approximation issues, *e.g.*, *false negative* in the verification processes. Others support data with ground variables which can entail the state space explosion problem.

The second challenge, which is related to the testing process, is the *limitation of controllability and observability* of choreography implementation which is a set of distributed services. Thus it is not easy to start or stop the services at testing moment. Furthermore, the full observations of overall implementation is usually not available. One can only observed partially the implementation at each participant service.

2.2 Choreography Modeling

We class the choreography models and languages based on some attributes which are depicted in Figure 2.6.

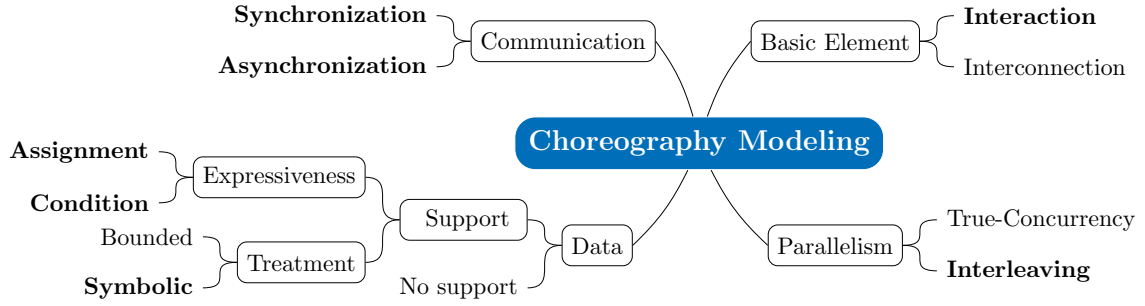


Figure 2.6: Criteria of Choreography Modeling

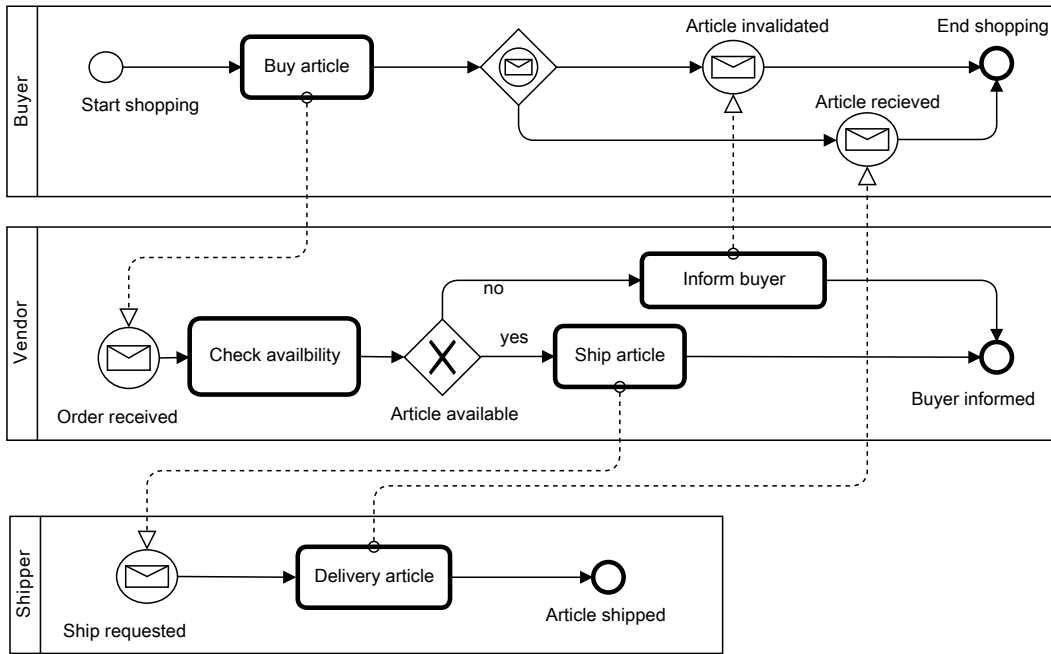
2.2.1 Basic Elements

Depending on the ways of definition of basic elements, there exist two different modeling approaches for choreographies [Decker et al., 2008]: interconnected interface models and interaction models. In *interconnected interface models*, the basic elements are the events defined at the role level (*e.g.*, sending or receiving a message). Global level interactions are then defined by roughly connecting these events. The messaging activities of different processes are interconnected. To the contrary, in *interaction models*, the basic elements are the interactions between roles. An interaction usually represents a message exchange between two participants in a choreography. Figure 2.7 demonstrates the two modeling approaches of Online-Shopping Process example.

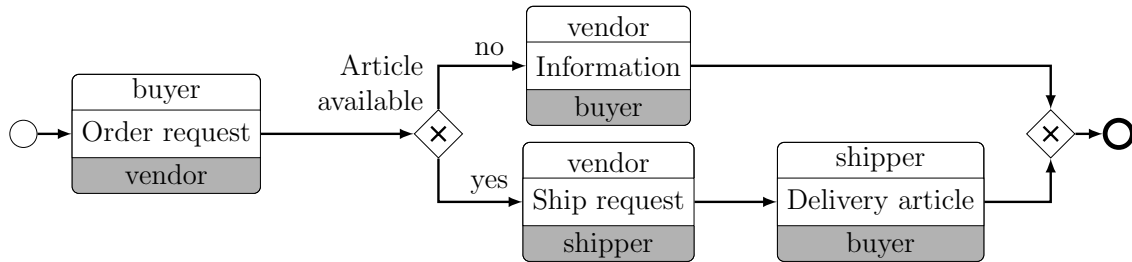
These two meta-models can be seen as different views of the same choreography. The interconnection model is suitable for implementation since it focuses on describing the activities of a participant service. Whereas the interaction model is suitable for specification and verification due to its global perspective.

Table 2.1 classifies choreography description languages into these two classes. WS-CDL was proposed by W3C firstly in 2005. Let's Dance [Zaha et al., 2006] is a graphical description of choreography. BPMN can only describe interconnection choreography in the first version. In version 2.0, it supports to describe interaction choreography. Collaboration Diagram (CD), also called communication diagram in Unified Modeling Language (UML) 2, are used to specify interaction choreography by [Bultan and Fu, 2008, Salaün et al., 2012].

Message Sequence Chart (MSC) can also describe choreographies [Foster et al., 2005]. It follows the interconnection modeling approach. However, it is rather suited for describing sequences of interactions in contrast to full choreographies. Complex branching, *e.g.*, parallel, conditional, is not well supported by MSCs [Decker and Weske, 2011]. Choreography Extension for BPEL (BPEL4Chor) is extended from WS-BPEL to specify interconnection models. ScriptOrc [Bhattacharjee and Shyama-



(a) Interconnection-based in BPMN 1.0



(b) Interaction-based in BPMN 2.0

Figure 2.7: Example of Interconnection and Interaction-based Modeling Approaches

[sundar, 2008](#)] is an algebra language for specifying interconnection choreography.

Table 2.1: Interaction *vs.* Interconnection in choreography languages

Interaction	Interconnection
– WS-CDL	– MSC
– Let’s Dance	– BPMN 1.0
– BPMN 2.0	– BPEL4Chor
– CD	– ScriptOrc

Table 2.2 classes choreography modeling approaches into three groups, Process Algebra-based, Automata-based, and Petri-net-based [Su et al., 2007]. Most works construct their models based on WS-CDL, *e.g.*, [Li et al., 2007a] proposed using Communicating Sequential Processes (CSP) to model WS-CDL.

Table 2.2: Interaction *vs.* Interconnection in Choreography Models

	References	Interaction	Language
Process Algebras	[Busi et al., 2006]	✓	WS-CDL
	[Kazhamiakin et al., 2006]	✓	
	[Li et al., 2007a]	✓	WS-CDL \mapsto CSP
	[Bravetti and Zavattaro, 2007]	✓	
	[Qiu et al., 2007]	✓	
	[Bhattacharjee and Shyamasundar, 2008]		ScriptOrc
	[Barker et al., 2009]		
	[Sun et al., 2010]	✓	WS-CDL
	[Salaün et al., 2012]	✓	CD \mapsto LOTOS
	[Yoon et al., 2011]	✓	
	[Poizat and Salaün, 2012]	✓	BPMN 2.0 Choreography \mapsto LOTOS
Automata	[Meng and Arbab, 2007]		
	[McIlvenna et al., 2009]	✓	
	[Mei et al., 2009]	✓	WS-CDL
	[Diaz and Rodriguez, 2009]	✓	
	[Zhou et al., 2010]	✓	WS-CDL
	[Hwang et al., 2010]		
Petri-nets	[van der Aalst et al., 2006]		
	[Decker and Weske, 2011]	✓	iBPMN
	[Valero et al., 2009]	✓	

An interaction model can be transformed to interconnection one and vice versa. In [Kopp et al., 2010], the authors present a mapping of interconnection models to interaction models by transforming BPMN 1.0 models into iBPMN models by using Petri-nets as intermediate format. [Decker and Weske, 2011] show a way to generate interconnection models out of interaction models.

2.2.2 Data Support

Abstract, Bounded and Unbounded Data Domain. Choreography modeling approaches differ on the way they deal with data. Some just *abstract away* from data. This may yield over-approximation issues, *e.g.*, *false negative* in the verification processes. Others support data with ground variables which can entail the state space explosion problem. For the following, let us suppose a very simple choreography C : “ A and B exchange an integer lower than 5”. Data can be supported by working on *closed implementation-level systems* where sent messages contain only ground data, *e.g.*, “ A sends 4 to B ” or “ A sends $x+3$ to B , knowing that x is 1”. In such a case, the state space explosion of the system model is limited. This is because even if the reception of a message in some entity is denoted with a free variable, *e.g.*, “ B receives from A a y lower than 10”, upon making it correspond with a sent message, the variable will be bound. Here, $y = 4$, satisfying B since $4 < 10$ and conforming to C since $4 < 5$. However, this is not adequate when working on abstract specifications where there are no such ground sent messages but only free variables and constraints on their values, *e.g.*, “ A sends some x greater than 3 to B ” (note, here, that the *exact* x is not given, since it can be known only at run time).

Another solution is to *bound data domains*, *e.g.*, integers are bound to $[0..b]$. The issue is that conformance may not yield outside the bounds. On our example, it works if $b < 5$ but not if $b \geq 5$ since A may then choose to send 6. Defining bounds in order to avoid false positives in the verification process can be difficult.

Data is supported using a *symbolic approach*, *i.e.*, the data variables are manipulated by using symbols rather than their concrete values. This enables one to model and analyze service choreography in presence of data without suffering from state space explosion and without bounding data domains.

The data support in choreography modeling approaches are compared in Table 2.3. In [Kazhamiakin and Pistore, 2006b, Knuplesch et al., 2012], the authors discuss about using symbolic model checking to analyze choreography but their choreography models still bound value domain of variables.

Data-Awareness Interaction. When data is supported, the basic element (first-class) of interaction models should be data together with interaction, called *data-awareness interaction*. The models used in [Xiangpeng et al., 2006, Busi et al., 2006, Kazhamiakin and Pistore, 2006b, Li et al., 2007b, Sun et al., 2010] are interaction-based but data are expressed as the way of interconnection-based, *i.e.*, the variables are defined at each local role then their values are synchronized thanks to interactions. This is not adequate for the interaction-based approach. The interactions together with variables should be the basic events. In the choreographies of Figure 2.9, variable

Table 2.3: Data Support in Choreography Modeling Approaches

	support	data-aware interaction	treatment	loops	assign.
[Qiu et al., 2007]	no			yes	no
[Bravetti and Zavattaro, 2007]				yes	no
[Salaün et al., 2012]				yes	no
[Diaz and Rodriguez, 2009]				yes	no
[Xiangpeng et al., 2006]	yes	no	closure	yes	yes
[Busi et al., 2006]				no	yes
[Li et al., 2007b]				yes	yes
[Sun et al., 2010]				no	yes
[Kazhamiakin and Pistore, 2006b]		yes	bound data	yes	no
[Knuplesch et al., 2012]				no	no
Ours-[Nguyen et al., 2012a]			symbolic	yes	limited

x is attached to the interaction o_1 , in which one does not need to specify explicitly that the first condition $x > 0$ must be done at a and the second one $x \leq 0$ at c .

Expressiveness. Column 5 and 6 of the Table 2.3 are relative to the expressiveness of the choreography language. Having both loops and assignments may yield state space explosion if one does not close the system or bound data domains. This can be avoided by working on closed implementation-level systems. Here, since sent messages contain only ground data (values), the state space explosion of the system is limited: even if the reception of a message is denoted with a free variable, upon making it correspond with a sent message, the variable will be bound. However, this is not adequate when working on abstract specifications where there are no such ground sent messages but only free variables and constraints on their values. Another solution is to bound data domains, *e.g.*, conformance may not yield outside the bounds, so choosing them is difficult.

2.2.3 Communication Modes

At global level, an interaction α is a single event. When α is projected on local models, it becomes two separate events: a sending, $\alpha!$, and a reception, $\alpha?$. The causality of the two local events depends on which communication model is considered. The *synchronous communication* requires that communication between events $\alpha!$ and $\alpha?$ can take place only if both are performed simultaneously, denoted as $\alpha! = \alpha?$. The *asynchronous communication* allows communication between events $\alpha!$ and $\alpha?$ to be consistent with the reception $\alpha?$ being performed after the sending $\alpha!$, denoted as $\alpha! \prec \alpha?$.

The causality of two consecutive interactions, $\alpha_1 \prec \alpha_2$, is considered at local level by the causality of their sending and reception. There exist four possibilities [Lanese

et al., 2008, Nguyen et al., 2012c]:

- *Synchronous communication*: $(\alpha_1! \prec \alpha_2!) \vee (\alpha_1? \prec \alpha_2?) \vee (\alpha_1! \prec \alpha_2?) \vee (\alpha_1? \prec \alpha_2!)$
- *Asynchronous-sending communication*: $\alpha_1! \prec \alpha_2!$
- *Asynchronous-reception communication*: $\alpha_1? \prec \alpha_2?$
- *Asynchronous-disjoint communication*: $\alpha_1? \prec \alpha_2!$.

Consequently, interaction models assume only synchronous communication [Kopp et al., 2010], since its communications are atomic event. The asynchronous communication modes are only considered when composing local models (of roles or services). In reality, asynchronous communication is usually achieved from synchronous communication by exchanging messages between senders and receivers through a certain communication medium, represented as a set of queues. This medium is referred as communication model in [Kazhamiakin et al., 2006, Kazhamiakin and Pistore, 2006b], as conversation protocol in [Fu et al., 2004, Bultan et al., 2006]. In asynchronous communication mode, the composition of local models depends on the kind of queues, *e.g.*, First In, First Out (FIFO), and on the size of queues. Hence some features of choreography, *e.g.*, realizability, conformance, depend also on these elements [Kazhamiakin and Pistore, 2006a, Salaün and Roohi, 2009, Salaün et al., 2012, Roohi and Salaün, 2011, Basu et al., 2012]. A composition of set of services is *synchronizable* [Basu and Bultan, 2011] if and only if the ordering of message exchanges remain the same when asynchronous communication is replaced with synchronous communication.

2.2.4 Parallelism

Since the choreography implementation is a set of services which are distributed processes, hence their activities might occur in parallel. Most approaches based on automata or on process algebra, interprets the parallel as interleaving, *i.e.*, two parallel events α_1 and α_2 are executed in sequence, either α_1 then α_2 or α_2 then α_1 . The true-concurrency means that events can occur at the same time. This attribute is well supported in approaches based on Petri-net.

2.3 Realizability & Projection

2.3.1 Realizability Notions

A fundamental issue of choreography, called *realizability*, is whether a choreography specification C can be implemented by an implementation $Impl$. Following [Kopp et al., 2010], realizability is a property of an interaction choreography model.

A trivial implementation of a choreography is a single service which plays all roles of the choreography. In such a case, there is no more realizability issue. However, choreography intends to specify, from a global view, a collaboration of a set of roles. Each role in the choreography is a concrete entity taking part in this collaboration. It should be implemented by a distinguishable, independent service. Consequently, an *implementation* of a choreography should be a set of services where each one implements one of the roles and their composition represents the behaviors required by the choreography.

Depending on how to say *Impl* implements *C*, *i.e.*, conformance relation, the realizability notions can be divided into three classes: complete, partial and distributed realizability as the following definitions. Given a choreography *C* which represents the collaboration of *n* roles, R_1, \dots, R_n , and an implementation *Impl* consisting of *n* services S_1, \dots, S_n . *Complete realizability* which is defined in [Bultan and Fu, 2008] requires that (i) each service s_i implements its roles R_i and (ii) the behaviors of the composition $S_1 \times \dots \times S_n$ are equals to the one of *C*. This is a strong requirement since it demands that behavior of services exactly matches the choreography. A weaker notion, called *partial realizability*, is defined in [Zaha et al., 2006]. It relaxes the second condition of the complete realizability by only requiring that a subset of the choreography is implemented. Thus the complete realizability implies the partial realizability. The first condition of these two notions demands each role is implemented by one and only one service. [Lohmann and Wolf, 2010] introduces *distributed realizability* which modifies this condition by allowing that a role can be realized by more than one services.

Without regarding on relation of pair role–service in the first requirement (i) above, realizability notion in [Kopp et al., 2010, Poizat and Salaün, 2012] only requires the second one. A realizable choreography can be implemented by a set of services where the composition exactly shows the specified message exchange behavior of the choreography. This definition is referred as *synchronous realizability* in [Kazhamiakin and Pistore, 2006a].

In reality, asynchronous communication is achieved from synchronous one by introducing a message queue at each participant service. Two notions are defined in [Basu et al., 2012] based on size of message queue. *Realizability_q* requires the existence of an implementation *Impl* such that its behaviors for all possible receive queue sizes are equivalent to the choreography *C*. *Realizability_∞* requires the existence of an implementation *Impl* such that its behavior is equivalent to the choreography *C* when unbounded receive queues are used.

2.3.2 Realizability through Projection

In a top-down service choreography approach, as depicted in Figure 2.4, service developers construct a global specification and project it into local specifications. Services are then selected to implement the local specifications. In this development approach, *the realizability issue becomes to verify whether a choreography model can be correctly projected onto role models*.

Projection is a procedure which takes as an input a choreography model with n roles and outputs a set of n local models, each one representing the required behaviors of a role in the choreography. Table 2.4 represents choreography projection approaches. To ensure the correctness of this projection, the conformance between the choreography model and the composition of the generated role models should be verified.

The realizability of choreography is checked in [Salaün et al., 2012, Roohi and Salaün, 2011, Poizat and Salaün, 2012] by firstly projecting the choreography onto local models. A system is built by composing these local models; some additional FIFO buffers are necessary if an asynchronous communication mode is assumed. Finally, the realizability is checked by verifying the equivalence of the choreography and the system.

Table 2.4: Service Choreography Projection Approaches

	Supporting Data	Smart Projection
[Qiu et al., 2007]	no	no (only support branching)
[Bravetti and Zavattaro, 2007]		no
[Salaün et al., 2012]		yes
[Diaz and Rodriguez, 2009]		yes
[Li et al., 2007b]	yes	no
[Xiangpeng et al., 2006]		no
[Sun et al., 2010]		yes
[Kazhamiakin and Pistore, 2006b]		no
[Knuplesch et al., 2012]		no
Ours-[Nguyen et al., 2013]		yes

2.3.3 Causes of Unrealizability

Choreography may often be not realizable. The reason of choreography unrealizability is that the interactions involve separate roles. Hence a role cannot compute itself which interactions it must or must not do. The role does the computation when it must perform an interaction after another interaction, or when it has to choose which branch must be followed. These two cases were illustrated in Figure 2.5. Two conditions of sequential composition and choice are only decided by one role [Qiu et al., 2007, Lanese et al., 2008]. It is called *dominant role* in [Qiu et al., 2007].

Another issue is that the realizability depends on the kind of communication modes, *e.g.*, synchronous or asynchronous communication. Figure 2.8 represents three choreographies which are realizable in synchronous communication mode but they are unrealizable in asynchronous ones. Let us consider the choreography in Figure 2.8(a) under asynchronous-sender communication mode where m_1 occurs before m_2 , *i.e.*, the choreography requires that the sending of m_1 must occur before the one of m_2 . Since there is no interaction between b and c , the services cannot respect the execution order of messages as specified in the choreography. The same reason of unrealizability holds for the two right choreographies.

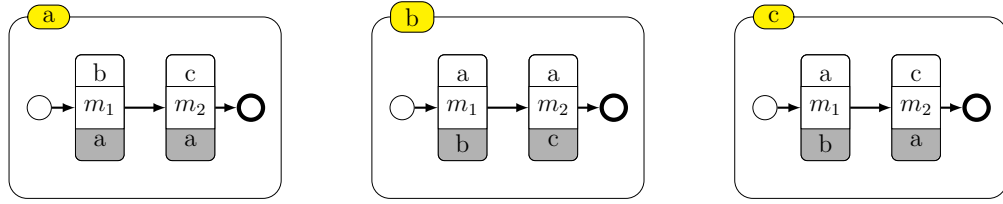


Figure 2.8: Impact of Communication Modes: Realizability in Synchronization \rightarrow Unrealizability in (a) Asynchronous-Sender, (b) Asynchronous-Receiver, and (c) Asynchronous-Disjoint

Few works focused on the realizability issue assuming asynchronous communication. Three sufficient conditions (lossless join, synchronous compatible, autonomous) were proposed in [Fu et al., 2004, Bultan et al., 2006] that guarantee a realizable conversation protocol. A service composition is *synchronous compatible* if its synchronous product does not contain a state in which there exists a service which has a send transition from its current local state, however, the corresponding receiver service is not in a state where it can receive that message. A service composition is *autonomous* if, given any state of any of the services, only one of the following three conditions hold 1) all the transitions from that state are send transitions, 2) all the transitions from that state are receive transitions, or 3) that state is a final state and there are no send or receive transitions from that state. A conversation protocol satisfies the lossless join condition if its conversation set is equal to the join of its projections to each service. Both necessary and sufficient conditions (equivalent and well-formed conditions) for realizability were provided by [Basu et al., 2012]. Before this work, the realizability of choreographies with asynchronous messages using unbounded FIFO message queues was undecidable.

2.3.4 Enforcing Realizable Choreographies

The unrealizability of choreography is usually caused by the lack of information which are used by participant to decide its behavior, *e.g.*, when to do the next behavior

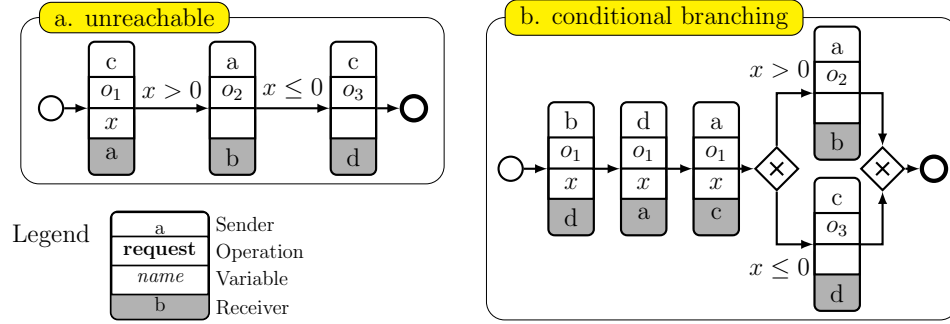
or which route needs to be followed. It is impossible to make an unrealizable choreography to a realizable one without introducing additional interactions [Qiu et al., 2007, Salaün et al., 2012, Diaz and Rodriguez, 2009, Sun et al., 2010]. Some approaches enable the realizability by privileging some roles in the choreography, *e.g.*, dominant role in [Qiu et al., 2007] or additional coordinators in [Diaz and Rodriguez, 2009, Autili et al., 2013]. These roles help the participants to decide their behaviors, *e.g.*, the route in the case of non-deterministic branching. The former approaches seem to be inappropriate in the sense of the choreography where there is no privileged roles.

As illustrated in Figure 2.8 where the choreographies are unrealizable under asynchronous communication mode but are realizable under synchronous one, hence some approaches try to make synchronizable choreography, [Salaün et al., 2012], or to reorder message emissions [Kazhamiakin and Pistore, 2006a].

Some other works, [Qiu et al., 2007, Basu et al., 2012] propose well-formed conditions to enforce designers to describe realizable choreographies. This approach is too restrictive and complex for designing choreographies since it may prevent the designer from specifying what (s)he wants to do. Furthermore, the designer cannot only focus on composition issues, but has to take care, at the same time, these well-formed conditions [Salaün et al., 2012]. This is contrary to our work of trying propose a realizable choreography based on an arbitrary one given by the designer.

2.3.5 Impact of Data on Realizability

Most existing approaches do not adequately support the value-passing, *i.e.*, without explicitly considering the data exchanged through interactions and using it for branching decisions. They just *abstract away from data*. This may yield *over-approximation issues*, *e.g.*, false negatives in verification process. Hence the presence of value-passing in choreography model may change its realizability property. Let us take an extension of the unrealizable choreographies in Figure 2.5 as depicted in Figure 2.9, which becomes realizable thanks to the value-passing. In this figure, the BPMN 2.0 notations are extended to support data [Knuplesch et al., 2012], hence an interaction is attached with a variable. The left choreography describes that firstly there is an interaction o_1 from role a to role c with data exchange called x . Then the interaction o_2 from a to b should be done if $x > 0$. Finally, the interaction o_3 from c to d should be done if $x \leq 0$. It is straightforward to see that the interaction o_3 is never executed. Hence there is no need of o_3 in the choreography. The choreography becomes realizable when only o_1 and o_2 are required to be implemented. The right choreography is realizable as well, when the communication is synchronous or on an asynchronous sending mode. After o_1 has been done, all roles know x so they can decide to do (send or receive) o_2 or o_3 .

Figure 2.9: Impact of Data: unrealizability \rightarrow realizability

The choreography modeling and realizability checking are also presented in [Paci et al., 2008, Bultan et al., 2009, Poizat and Salaün, 2012, Basu et al., 2012]. The realizability is checked in [Paci et al., 2008] based on access control policies of local services and the credentials that local services are willing to disclose. [Bultan et al., 2009, Poizat and Salaün, 2012] propose tools for checking the realizability of choreography modeled by collaboration diagrams and by BPMN 2.0. The necessary and sufficient conditions for realizability of choreographies are given in [Basu et al., 2012]. However, all these approaches do not explicitly support data exchanged by messages and used for routing decisions.

2.4 Conformance Checking

The second key issue in choreography-oriented development is checking the *conformance* of an implementation *wrt.* the choreography specification. When the model of the implementation is available, the conformance checking can be done, otherwise conformance testing will be done. However, it is generally desirable to detect faults as soon as possible in system development.

2.4.1 Choreography Conformance

The conformance issue naturally arises in a bottom-up development process, where peers implemented as services and advertising behavioral descriptions (conversations) are reused and composed. The question here is “*Will these services behave altogether as prescribed in the choreography?*”. Conformance is also central in a top-down development process, see Figure 2.4, where local obligations, kinds of “behavioral skeletons”, are generated by projection from the choreography. The question here is “*Am I sure that if I implement these local obligations then I will have a correct implementation of the choreography?*”. Conformance is therefore a cornerstone for

choreography realizability checking that addresses whether projection may or may not be used to get a conform set of services skeletons.

Since choreography conformance intends to check whether an implementation of a choreography conforms to its specification, the first notion we need to define is an implementation of a choreography. As discussion in Section 2.3.1, *an implementation of a choreography* is a set of coordination services, each services implements a role in the choreography and the coordination between them conforms to the choreography specification. With such definition of implementation, choreography conformance consists of local and global conformance. *Local conformance* ensures behavior of each service respects to its role. *Global conformance* guarantees the collaboration of the set of services respects to the choreography specification.

2.4.2 Conformance Relation

Conformance is a binary relation between an implementation model *Impl* and a specification *Spec*.



Figure 2.10: Conformance Checking Scheme

The conformance is usually a non symmetric relation since it allows *Impl* to be less than *Spec*. If we note $\text{behavior}(X)$ the set of all potential behaviors of X , then we have the following equation:

$$\text{behavior}(Impl) \stackrel{\subseteq}{=} \text{behavior}(Spec)$$

For example, the conformance relation stated by [Su et al., 2007] does not allow behaviors of *Impl* to be larger than behaviors of *Spec*:

“Assuming some fixed choreography modeling language \mathcal{L} : Given a choreography C in \mathcal{L} and a set I of service implementations in \mathcal{L} , is it possible to determine if *every possible execution of I always generates the behaviors allowed by C* ?”

In Table 2.5, we compare approaches for conformance checking. Column 2 focuses on data support.

Table 2.5: Choreography Conformance Relations

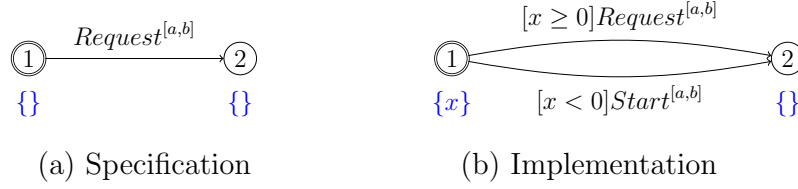
	Data supported	Conformance		
		local	global	relation (based on)
[Qiu et al., 2007]	no	no	yes	Trace equivalence
[Bravetti and Zavattaro, 2007]		yes	yes	Trace equivalence
[Salaün et al., 2012]		no	yes	Strong bisimulation
[Basu and Bultan, 2011]		no	yes	Pre-Order
Ours [Nguyen et al., 2012c]		yes	yes	Pre-Order
[Busi et al., 2006]	yes	no	yes	Branching bisimulation
[Kazhamiakin and Pistore, 2006b]		no	yes	Branching bisimulation
[Li et al., 2007b]		yes	no	Weak bisimulation
Ours [Nguyen et al., 2012a]		no	yes	Branching bisimulation

The three last columns of Table 2.5 ingredient is how to formulate the behaviors of an execution of services and compare them against a choreography. One straightforward approach is to use traces of the service executions modulo irrelevant events. This has been mostly studied in automata based choreography models. The other is to employ the notion of bisimulation between the generated global behaviors and choreography. Most process algebra based choreography models adopt this approach. Weak and branching bisimulations are able to support internal events and hiding (formally, τ actions). Branching bisimulation [Van Glabbeek and Weijland, 1996] has been preferred over weak bisimulation in the last years since it is a congruence, hence supports compositional reasoning.

Most approaches allows the implementation to have some irrelevant behaviors which will be removed or hidden in the conformance checking process. This is important, *e.g.*, if one has to deal with messages added to make some choreography realizable.

2.4.3 Impact of Data on Conformance Relation

Most approaches do not take data into account their conformance relation. Let us investigate the impact of data on conformance relation by considering example in Figure 2.11. This figure represents two models. The left one specifies a very simple choreography consisting of two roles, *a* and *b*, and an interaction *Request* from role *a* to role *b*. The right one describes the collaboration of an implementation of the choreography. The implementation can do from role *a* to role *b* a *Request* interaction or a *Start* interaction depending on value of the variable *x* at the initial state, *i.e.*, this value is configured before running. Hence the implementation conforms to the choreography if and only if *x* is configured greater or equal 0, *e.g.*, when we examine only nature numbers. Consequently, the conformance checking is neither **Pass** nor **Fail**. In such a case, we issue **Maybe** verdict, see Section 3.3.

Figure 2.11: Impact of Data on Conformance Relation: **Maybe** Verdict

2.5 Choreography Testing

2.5.1 Formal Software Testing

Software testing is the process of executing a system with the intent of finding faults [Glenford J. Myers, 2004]. The standard definition was presented in IEEE Standard Glossary of Software Engineering Terminology [Jane Radatz, 1990]:

“Test is an activity in which a system or component is executed under specified conditions, the results are observed or recorded, and an evaluation is made of some aspect of the system or component.”

Specification-based testing aims to test the functionality of an implementation, called IUT, according to a requirement specification. A *correct* implementation of a specification will contain no faults. Therefore, it can be achieved by either constructing it without making any errors, or by detecting and removing all the faults. The definitions of the failure, the fault and the error are relative among authors. In [Laycock, 1993], the authors represented these notions in the relation between a specification and its implementation.

Given an implementation $Impl$, of a specification $Spec$, a *failure* occurs if for an input i , the output produced by $Impl$ is unacceptable compared to the output produced by $Spec$. Any part of the system state, that could lead to failures, is a *fault*. An *error* is the direct cause of a fault. In the case of a program fault, the error is often a mistake by the programmer.

With these definitions, the test can be redefined by the conformance relation between the implementation and its specification.

Verdicts. The execution of each test provides one of the three possible verdicts: **Pass**, **Fail** and **Inconclusive**. The **Pass** verdict means that the test does not

detect any faults. The **Fail** verdict means that the test detected a fault. The **Inconclusive** will be emitted when neither **Pass** nor **Fail** verdicts are emitted, *i.e.*, no definite conclusion can be drawn from the test. In conformance testing, an IUT conforms to a specification if there is no **Fail** verdicts for every test.

Testing vs. Proof of Correctness, Model Checking, Debugging. Testing is very different from proof of correctness and model checking. Testing works on an actual IUT. Whereas proof of correctness and model checking requires a model of the IUT, which is usually not available. Based on model of IUT, proof of correctness uses the technique of a formal logic system to prove that if the input values satisfy certain constraints, then the output values produced by the IUT, will satisfy certain properties. Meanwhile model checking verifies all potential behaviors of IUT with respect to the specification by regarding their models.

Testing is not only debugging. The purpose of testing can be also quality assurance, verification and validation, or reliability estimation. Debugging is only done at code stage whereas testing can be also done after the code stage, *i.e.*, when system is deployed.

Active and Passive Testing. Formal software testing can be differentiated as active or passive. *Active testing* consists in applying a set of test cases by sending some input to the IUT and then analyzing its output in order to emit a verdict. This method assumes a kind of controllability of the IUT through Point of Control and Observation (PCO).

To the contrary of active testing which has the controllability, the passive testing does not disturb the natural operation of the IUT. In the *passive testing* [Lee et al., 1997], the tester can not send messages to the IUT. It only observes the exchange (input and output) of messages, through the PO, of an IUT during run-time. These observations will then be compared to the specification in order to emit a verdict. Consequently, it is also of particular interest since one does not always have the ability to control an IUT. By using passive approach, testing can be done continuously and the services in a collaboration can evolve dynamically.

Following [Ladani et al., 2005], passive testing approach is divided into two groups, naïve and property-oriented based approaches. The *naïve based approach* consists in comparing the specification trace with the one of the implementation in a forward or backward manner. Whereas in *property-oriented based approach*, as shown in Figure 2.12(b), only critical properties, which are given either by an expert or extracted from the standard, are compared with the execution trace [Bayse et al., 2005]. If specification is available, then the properties must conform to the specification.

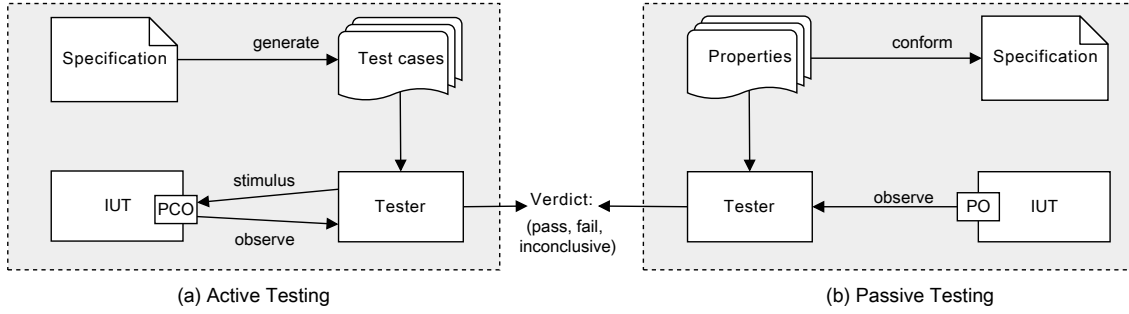


Figure 2.12: Active and Passive Testing Processes

Black, White and Grey Box Testing. *Black-box testing*, called also *functional testing*, treats the IUT as black-box, *i.e.*, the tester does not have any internal views of the IUT. Contrary to black-box testing, in *white-box testing*, IUT is viewed as a white-box, or glass-box, *i.e.*, the structure and flow of the IUT are visible to the tester. *Grey-box testing* is a common used transition between black-box and white-box testing. Testers have access to the system source code, but testing process is usually executed as the black-box [Repasi, 2009].

Controllability, Observability and Testability. *Controllability* is the degree to which it is possible to control the state of the IUT as required for testing. *Observability* is the degree to which it is possible to observe (intermediate and final) test results. For instance, in active testing, the communication between the test system and the IUT are realized by the input or output messages at PCOs in a given order. In this context, the *controllable* message is the messages which are sent by tester and the *observable* message the messages received by tester [Rafiq and Cacciari, 2003].

The controllability and observability have a great influence on several aspects of the testing activity, for example, the execution of test sequences, the observations that can be made during the test execution. The *testability* is the degree to which a IUT facilitates the establishment of test criteria and the performance of testing process [Jungmayr, 2004].

2.5.2 Service Choreography Testing

In Table 2.6, we compare approaches for choreography testing. The criterion are detailed as below.

Passive Testing. The first criteria is passive testing. Each service in a choreography implementation is usually developed and deployed independently with the other services. Furthermore, lack of controllability of a distributed system prevents active

Table 2.6: Service Choreography Testing Approaches

	Passive		Testing architecture	Value-Passing		Online	Global clock
	sup.	prop.		support	structure		
[Zhou et al., 2010]	no	no	centralized	yes	no	no	yes
ours-[Nguyen et al., 2012c]	yes		centralized	no	no	no	yes
[Andrés et al., 2010]		yes	distributed	no	no	no	no
[Hallé and Villemaire, 2009]			centralized	yes	no	yes	yes
ours-[Nguyen et al., 2012b]			distributed	yes	yes	yes	no

testing. Passive testing becomes a potential candidate for testing service choreography in which the tester discovers the behaviors of a service only by observing from outside its inputs and outputs. Column 3 is relative to property-oriented passive testing approaches which allows testers to focus on some critical behavior of IUT.

Architecture of Passive Testing System. The second criteria is based on testing architecture. Testing architecture is a description of the behavior of testers by which the IUT is tested. A choreography implementation can be tested by many testers at different sites of the implementation. The testing architecture is distinguished by terms of coordination between the testers in the testing process. There exists three kinds of testing system.

- *Centralized testing* consists of only one tester.
- *Remote testing* consists of several testers located at different sites. There is no coordination between the testers, and each of them behaves with its corresponding PO, independently of the others.
- *Distributed testing* consists of different testers like in remote testing, but there is a coordination between the testers. If this coordination has to follow given specific procedures, it is called a coordinated testing.

The basic idea is to coordinate the testers by using a communication service parallel to the IUT through a multicast channel. Each tester observes the IUT only through the port PO to which it is attached, and communicates with the other testers through the multicast channel. One tester dose not perform only in the observations with the IUT through the attached port PO, but also the coordination messages with the other testers while the testing is carried out.

Data Support. Columns 5 and 6 of the Table 2.6 are relative to the data support in testing process. Some approaches, [Andrés et al., 2009, 2010, Nguyen et al., 2012c] just *abstract away from data*. This is known to yield over-approximation issues, *e.g.*,

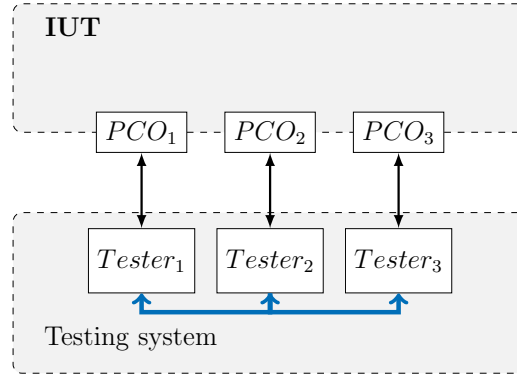


Figure 2.13: An Architecture of Distributed Testing

false negatives in the verification process. Furthermore, the information exchange is usually in complex type, thus complex data treatment should be supported in testing.

Online Testing. Column 7 of the Table 2.6 distinguishes online and offline realization testing approaches. Testing process is done to detect a fault in IUT. In some case, it is intended to be able to detect faults as soon as they are generated by the IUT, thus the IUT can be prevented to go to serious errors, *e.g.*, damage of material. It is the case of online testing. Contrarily, offline testing performs the test after IUT terminating its execution.

Global clock. In distributed testing, each tester knows only its attached service. It has only (local) observations about this service, thus partial observations about the choreography. To test global conformance, tester needs to correlate local observations to construct global observation. The correlation is usually done based on time stamp of local observations. Consequently, a global (logic) clock or a shared clock is required for a choreography implementation, *i.e.*, at a moment, all local clocks of testers indicate the same value. This requirement is usually not achieved due to the distributed environment.

A Symbolic Model of Choreographies

Contents

3.1	Specification Language	36
3.2	Symbolic Semantics	39
3.3	Symbolic Conformance	43
3.4	Realizability Checking & Projection	51
3.5	Tool & Experimental Evaluations	58
3.6	Discussion	65

In this chapter, we present a framework to model and analyze service choreographies. Particularly, we first introduce our formal language and model for interaction-based service choreographies with value-passing. The symbolic semantics of the model are also given. Based on the proposed formal model, we then examine three fundamental issues of choreography, *e.g.*, (global) conformance, projection and realizability. Finally, we present the validation of our approach by our tools.

3.1 Specification Language

One important motivation in setting up the set of activities that can be specified in our language is BPMN, the standard notation for business process modeling that is able to support in its 2.0 version both orchestration and choreographic modeling. We plan to define model transformation from BPMN 2.0 to our language in the future, thus enabling the seamless integration of the verification techniques we describe here in BPMN 2.0 modeling tools.

Recently, process algebras have been receiving attentions in developing choreography modeling by giving a formal semantics to business process languages and notations [Su et al., 2007]. We take inspiration in all-purpose process algebras such as LOTOS and choreography-oriented ones [Qiu et al., 2007, Bravetti and Zavattaro, 2007].

3.1.1 Basic Events: Interaction *vs.* Sending & Reception

In interaction-based model, the basic event in choreography is an interaction. An interaction represents a communication between two roles. In this section, we will extend interactions with data. We distinguish two kinds of interactions: free interactions and bound interactions.

Let $a, b, \dots \in \mathcal{R}$: be a finite set of roles,
 $o, o_1, \dots \in \mathcal{O}$: be a finite set of operations, and
 $x, y, z, x_1, \dots \in \mathcal{V}$: be a finite set of variables,

a *free interaction* represents a communication of value of variable x realized through an operation o from role a to b is denoted by $o^{[a,b]}.x$, while the bound one is denoted by $o^{[a,b]}. \langle x \rangle$.

The difference between free and bound interaction relies on how their variables representing data exchange are interpreted, *i.e.*, the variables are free or bound, see their semantics defined in Figure 3.1. Particularly, in free interaction, the data exchange must be known before the interaction may occur, *e.g.*, in this free interaction, $o^{[a,b]}.x$, value of x was established somewhere before occurrence of the interaction, hence this interaction transfers the value of x from a to b through operation o . Otherwise, in bound interaction, the data exchange is bounded at the moment the interaction occurs, *e.g.*, when this bound interaction, $o^{[a,b]}. \langle x \rangle$, happens, the variable x is assigned a new value, hence its old value is overridden, and the new value is transferred from a to b through operation o . We remark that x may be a complex structure variable. The variable x may be omitted when the interaction does not carry data, *e.g.*, $\text{ACK}^{[a,b]}$, or when it is never used.

Table 3.1: The Basic Events

α	Kind	Local/Global	Free/Bound	$\mathbf{fv}(\alpha)$	$\mathbf{bv}(\alpha)$
$o^{[a,b]}.x$	Free Interaction	global model	f	$\{x\}$	\emptyset
$o^{[a,b]}. \langle x \rangle$	Bound Interaction		b	\emptyset	$\{x\}$
τ	Silent	local model	f	\emptyset	\emptyset
$o^{[a,b]?} \langle x \rangle$	(Bound) Reception		b	\emptyset	$\{x\}$
$o^{[a,b]}!x$	Free Sending		f	$\{x\}$	\emptyset
$o^{[a,b]}! \langle x \rangle$	Bound Sending		b	\emptyset	$\{x\}$

At local views, we introduce a (bound) reception $o^{[a,b]?} \langle x \rangle$, a free sending $o^{[a,b]}!x$, and a bound sending $o^{[a,b]}! \langle x \rangle$. They are used to represent events of each participant in choreography. Table 3.1 lists our basic events, in which we define $\mathbf{fv}(\alpha)$ and $\mathbf{bv}(\alpha)$ as the set of free and bound variables in α . The semantics of these events respect the late symbolic semantics of the free and bound events as presented in [Hennessy and Lin, 1995].

In the Table 3.1, τ event is considered for local model. In the literature, τ events are usually used to represent unobservable events, *e.g.*, internal events. Since we are interested in an abstract formal choreography model, *i.e.*, implementation independent and in interaction-based model, the basic events are interactions, the internal event can be ignored at choreography level [Kopp and Leymann, 2009]. Although there are no internal events in our approach, τ is used to represent an unobservable interaction. Consequently, it does not exist at global model but may appear on local models to express interactions of third participants, *e.g.*, these interactions do not concern with the current participant, this is why they are not observed.

3.1.2 General Language

The syntax of our specification language, \mathcal{L} , is given by:

$$\mathcal{L} ::= \mathbf{1} \mid \alpha \mid \mathcal{L}; \mathcal{L} \mid \mathcal{L} + \mathcal{L} \mid \mathcal{L} | \mathcal{L} \mid \mathcal{L} [> \mathcal{L} \mid [\phi] \triangleright \mathcal{L} \mid [\phi] * \mathcal{L}$$

In \mathcal{L} we can specify activities which are either basic or composite. A basic activity is either an inaction ($\mathbf{1}$), or a standard basic event (α) presented above. We then have structuring operators, that can be used to specify composite activities such as sequencing ($;$), non-deterministic choice ($+$), parallel activities ($|$), and interruption ($[>]$). Furthermore, since data exchange is supported, we have data-based conditional constructs, namely guards (\triangleright) and while loops ($*$), where ϕ is a condition (a boolean expression). The decreasing priority of operators is \triangleright and $*$, $;$, $+$, $|$, and $[>]$.

Choreography (Global) Specification. A choreography specification describes, with a global perspective, the legal interactions among roles played by the participants of a distributed system. In a choreography, each role is identified by a unique name. A *choreography (or global) specification* for a set of roles $R \subseteq \mathcal{R}$, a set of operations $O \subseteq \mathcal{O}$, and a set of variables $V \subseteq \mathcal{V}$, is an element of \mathcal{L} with the basic event α is either a free interaction or a bound interaction, *i.e.*, $\alpha \in \{o^{[a,b]}.x, o^{[a,b]}. \langle x \rangle \mid o \in O \wedge a \in R \wedge b \in R \wedge a \neq b \wedge x \in V\}$.

Example 1 (Online-Shopping). *Let us suppose a very simple Online-Shopping choreography between two roles: b (buyer) and v (vendor). The buyer first requests an article by providing an amount to be bought. If the amount is greater than 5 then the vendor aborts this transaction. Otherwise, a confirmation will be issue from the vendor to the buyer. This can be described as follows:*

$$C_1 ::= \text{Request}^{[b,v]}. \langle x_1 \rangle; ([x_1 < 5] \triangleright \text{Confirm}^{[v,b]} + [x_1 \geq 5] \triangleright \text{Abort}^{[v,b]})$$

Role and Service (Local) Specification. In a top-down development process, local descriptions correspond to role requirements (or role for short) obtained by projection from global specifications. Each role describes what is expected from a service that would implement it. In a bottom-up development process, local specifications correspond to the behavioral contracts or interfaces that services advertise in order to foster reuse, composition, and adaptation [Poizat, 2011]. In the sequel, we will use the term *local entity*, or *entity* for short, in order to abstract from the process development being used. An *entity (or local) description* for an entity a , a set of roles $R \subseteq \mathcal{R}$ with $a \in R$, a set of operations $O \subseteq \mathcal{O}$, and a set of variables $V \subseteq \mathcal{V}$, is an element of L with the basic event α is either a free sending, or a bound sending, or a reception, *i.e.*, $\alpha = \{o^{[a,b]}!x, o^{[a,b]}! \langle x \rangle, o^{[b,a]}? \langle x \rangle \mid o \in O \wedge b \in R \wedge a \neq b \wedge x \in V\}$.

Example 2. *A tentative to implement the Online-Shopping choreography (which is correct as we will see in the following) is that the buyer sends the amount of the article to the vendor and then waits for either abort or confirmation, while it is the vendor that checks the amount in order to give its decision:*

$$\begin{aligned} \text{Buyer } b &::= \text{Request}^{[b,v]}! \langle y_1 \rangle; (\text{Abort}^{[v,b]}? + \text{Confirm}^{[v,b]}?) \\ \text{Vendor } v &::= \text{Request}^{[b,v]}? \langle z_1 \rangle; ([z_1 < 5] \triangleright \text{Confirm}^{[v,b]}! + [z_1 \geq 5] \triangleright \text{Abort}^{[v,b]}!) \end{aligned}$$

3.2 Symbolic Semantics

3.2.1 Symbolic Transition System

We use Symbolic Transition Graphs (STG) [Hennesy and Lin, 1995] as a formal model for service choreographies. It can be used to specify either global view or local view by using global or local events. We select STG as model for service choreography since it supports data, guard and free/bound variables [Hennesy and Lin, 1995, Van Glabbeek and Weijland, 1996, Deng and Lin, 2005]. Many formal approaches and notably conformance relation are based on such a model [Pathak et al., 2008]. Moreover, Symbolic Transition systems, which is very close to STG, are widely used in different areas, *e.g.*, for testing purpose [Gaston et al., 2006, Bentakouk et al., 2009], or for code generation [Pavel et al., 2005, Fernandes and Royer, 2007, Fernandes et al., 2007].

A STG is a transition system. Each transition of STG is labelled by a guard ϕ and a basic event α . The guard ϕ is a boolean equation which has to hold for the transition to take place. A symbolic transition from state s to state s' with a guard ϕ , and an event α is denoted as $s \xrightarrow{[\phi]\alpha} s'$. The guard ϕ of a transition can be omitted if it is true, *e.g.*, $s \xrightarrow{\alpha} s'$. We add a specific event, \checkmark , to denote activity termination.

Definition 1 (Symbolic Transition Graph). *Formally, a STG is a tuple (S, s_0, T) where, S is a non empty set of finite states, each state s having an associated set of free variables $\text{fv}(s)$ which are used by guards or events in next transitions, $s_0 \in S$ is the initial state, and T is a set of finite transitions.*

If $s \xrightarrow{[\phi]\alpha} s'$ is a transition of T then $\text{fv}(\phi) \cup \text{fv}(\alpha) \subseteq \text{fv}(s)$ and $\text{fv}(s') \subseteq \text{fv}(s) \cup \text{bv}(\alpha)$.

A non-symbolic semantics for STGs would bound free variables using domain enumeration, *e.g.*, bound an integer x to $\{0, 1, 2, \dots\}$, thus yielding state space explosion. In the symbolic semantics, substitutions are associated to STG states. Let us begin with some additional notations.

A *variable substitution* (substitution for short) σ is a mapping from \mathcal{V} to \mathcal{V} . \emptyset denotes the empty substitution. $\sigma[x \mapsto y]$ is the substitution σ where the mapping from x to y is added, eventually erasing a previous mapping from x . $e\sigma$ denotes the application of σ to e . A *term* [Hennesy and Lin, 1995] is a pair s_σ where s is a state and σ is a substitution. In the sequel, we denote states by s, s_1, s' , etc. and terms by t, t_1, t' , etc.

We may now define the (late) symbolic semantics of an STG as relation over terms

(Figure 3.1). Under late semantic, for every transition $s \xrightarrow{[\phi]\alpha} s'$, the free variable in ϕ and α will be changed by the substitution of s . Moreover, if α is a bound event, e.g., $o^{[a,b]}. \langle x \rangle$, $o^{[a,b]}? \langle x \rangle$, or $o^{[a,b]}! \langle x \rangle$, then the substitution of s' is the one of s by updating a substitution from x to a fresh variable, z , instead of a value. Please note that \rightarrow is used for STG transitions (over states) and \mapsto is used for STG semantics (over terms).

$$\begin{array}{ccc}
\text{(TAU)} & \text{(FREE)} & \text{(BOUND)} \\
\frac{s \xrightarrow{[\phi]\alpha} s'}{s_\sigma \mapsto^{[\phi\sigma]\alpha} s'_\sigma} & \frac{s \xrightarrow{[\phi] o^{[a,b]}. \bullet x} s'}{s_\sigma \mapsto^{[\phi\sigma] o^{[a,b]}. \bullet (x\sigma)} s'_\sigma} & \frac{s \xrightarrow{[\phi] o^{[a,b]} \circ \langle x \rangle} s'}{s_\sigma \mapsto^{[\phi\sigma] o^{[a,b]} \circ \langle z \rangle} s'_\sigma} \\
\text{with } \alpha \in \{\tau, \checkmark\} & \bullet \in \{!, .\} & \circ \in \{?, !, .\}
\end{array}$$

Figure 3.1: Semantics of STG

3.2.2 Transformation Rules

We use STGs as a formal model to give semantics to our language. This is achieved by a rule-based transformation defined in Figure 3.2 where $\mathbf{0}$ is the empty description (only used for semantics), \checkmark denotes activity termination, α denotes any event but for \checkmark , and $\hat{\alpha}$ denotes any event (including \checkmark). The symmetric rules for CHOICE_1 and PAR_1 can be inferred from them and are omitted here.

$$\begin{array}{ccc}
\text{(SKIP)} & \text{(ACT)} & \text{(SEQ}_1 - L_1 \text{ does not end)} \quad \text{(SEQ}_2 - L_1 \text{ ends, begin } L_2) \\
\frac{}{\mathbf{1} \xrightarrow{[true]\checkmark} \mathbf{0}} & \frac{}{\alpha \xrightarrow{[true]\alpha} \mathbf{1}} & \frac{L_1 \xrightarrow{[\phi]\alpha} L'_1}{L_1; L_2 \xrightarrow{[\phi]\alpha} L'_1; L_2} \quad \frac{L_1 \xrightarrow{[\phi_1]\checkmark} L'_1, L_2 \xrightarrow{[\phi_2]\hat{\alpha}} L'_2}{L_1; L_2 \xrightarrow{[\phi_1 \wedge \phi_2]\hat{\alpha}} L'_2} \\
\\
\text{(CHOICE}_1 - \text{choose } L_1) & \text{(PAR}_1 - \text{one step in } L_1) & \text{(PAR}_3 - \text{synchronous termination)} \\
\frac{L_1 \xrightarrow{[\phi]\hat{\alpha}} L'_1}{L_1 + L_2 \xrightarrow{[\phi]\hat{\alpha}} L'_1} & \frac{L_1 \xrightarrow{[\phi]\alpha} L'_1}{L_1 | L_2 \xrightarrow{[\phi]\alpha} L'_1 | L_2} & \frac{L_1 \xrightarrow{[\phi_1]\checkmark} L'_1, L_2 \xrightarrow{[\phi_2]\checkmark} L'_2}{L_1 | L_2 \xrightarrow{[\phi_1 \wedge \phi_2]\checkmark} L'_1 | L'_2} \\
\\
\text{(GUARD)} & \text{(LOOP}_1 - \text{one more iteration)} & \text{(LOOP}_2 - \text{end of the loop)} \\
\frac{L \xrightarrow{[\phi']\hat{\alpha}} L'}{[\phi] \triangleright L \xrightarrow{[\phi \wedge \phi']\hat{\alpha}} L'} & \frac{L \xrightarrow{[\phi']\hat{\alpha}} L'}{[\phi] * L \xrightarrow{[\phi \wedge \phi']\hat{\alpha}} L'; [\phi] * L} & \frac{}{[\phi] * L \xrightarrow{[\neg\phi]\checkmark} \mathbf{0}} \\
\\
\text{(INTER}_1 - \text{one } L_1 \text{ step)} & \text{(INTER}_2 - L_1 \text{ ends, interruption not possible)} & \text{(INTER}_3 - \text{interruption by } L_2) \\
\frac{L_1 \xrightarrow{[\phi]\alpha} L'_1}{L_1 \triangleright L_2 \xrightarrow{[\phi]\alpha} L'_1 \triangleright L_2} & \frac{L_1 \xrightarrow{[\phi]\checkmark} L'_1}{L_1 \triangleright L_2 \xrightarrow{[\phi]\checkmark} L'_1} & \frac{L_2 \xrightarrow{[\phi]\hat{\alpha}} L'_2}{L_1 \triangleright L_2 \xrightarrow{[\phi]\hat{\alpha}} L'_2}
\end{array}$$

Figure 3.2: Transformation from our Language to STGs

3.2.3 STG Product

The product of STGs is used to give a semantics to a composition of a set of interacting local entities (Figure 3.3). We assume that the STGs use disjoint sets of variables which can be achieved using, *e.g.*, indexing by the name of the entity. The first rule (TERM) expresses that the composition terminates successfully when all its components do so. The second rule (PAR₁) represents the independent evolution of one of the STGs (the first one). The third rule (FREE-COM) denotes a synchronization between a sending a data in x to b and b receiving it in variable y . In this rule, $t[y \mapsto x]$ with $t = s_\sigma$ denotes the term $s_{\sigma[y \mapsto x]}$. Further, $\langle t_1, t_2 \rangle$, with $t_1 = s_{1\sigma_1}$ and $t_2 = s_{2\sigma_2}$, denotes the term $(s_1, s_2)_{\sigma_1 \cup \sigma_2}$. The symmetric rules for these rules can be inferred from them and are omitted here. We give a binary version of the product for simplicity purposes. An n -ary version of it can be obtained working on tuples $\langle t_1, \dots, t_n \rangle$ of terms instead of pairs $\langle t_1, t_2 \rangle$, generalizing TERM to n terminations, and having n symmetric rules PAR₁, ..., PAR _{n} . COM does not change being generic on a and b .

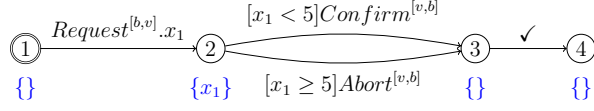
$$\begin{array}{c}
 \text{(TERM)} \\
 \frac{t_1 \vdash \xrightarrow{[\phi_1] \checkmark} t'_1 \quad t_2 \vdash \xrightarrow{[\phi_2] \checkmark} t'_2}{\langle t_1, t_2 \rangle \vdash \xrightarrow{[\phi_1 \wedge \phi_2] \checkmark} \langle t'_1, t'_2 \rangle}
 \end{array}
 \qquad
 \begin{array}{c}
 \text{(FREE-COM)} \\
 \frac{t_1 \vdash \xrightarrow{[\phi_1] \quad c[a,b]!x} t'_1 \quad t_2 \vdash \xrightarrow{[\phi_2] \quad c[a,b]? \langle y \rangle} t'_2}{\langle t_1, t_2 \rangle \vdash \xrightarrow{[\phi_1 \wedge \phi_2] \quad c[a,b].x} \langle t'_1, t'_2[y \mapsto x] \rangle}
 \end{array}$$

$$\begin{array}{c}
 \text{(PAR}_1\text{)} \\
 \frac{t_1 \vdash \xrightarrow{[\phi] \alpha} t'_1 \quad \alpha \in \{o^{[a,b].x}, o^{[a,b].\langle x \rangle}\}}{\langle t_1, t_2 \rangle \vdash \xrightarrow{[\phi] \alpha} \langle t'_1, t_2 \rangle}
 \end{array}
 \qquad
 \begin{array}{c}
 \text{(BOUND-COM)} \\
 \frac{t_1 \vdash \xrightarrow{[\phi_1] \quad c[a,b]!\langle x \rangle} t'_1 \quad t_2 \vdash \xrightarrow{[\phi_2] \quad c[a,b]? \langle y \rangle} t'_2}{\langle t_1, t_2 \rangle \vdash \xrightarrow{[\phi_1 \wedge \phi_2] \quad c[a,b].\langle x \rangle} \langle t'_1, t'_2[y \mapsto x] \rangle}
 \end{array}$$

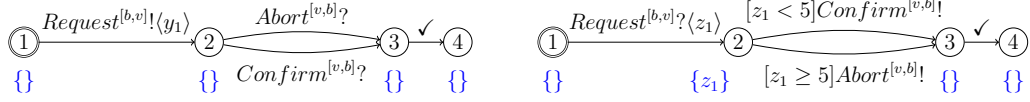
Figure 3.3: Rules for the Product of STGs

Example 3. The STGs for the choreography in Example 1 and for the buyer and vendor in Example 2 are shown in Figure 3.4(a-c). Figure 3.4(d) presents the product of the STGs in Figure 3.4(b) and Figure 3.4(c). The free variables of the states are given below them, *e.g.*, $\{x_1\}$ for state 2 in the choreography STG.

The Algorithm 1 represents the implementation of the product rules in Figure 3.3. In this algorithm, we use \sqcup to denote disjoint union, *i.e.*, $S_1 \sqcup S_2$ is defined only if $S_1 \cap S_2 = \emptyset$. We use also two other functions `renameVariable` and `updateFreeVariables`. The first one, `renameVariable(\mathcal{S}, s, v, v')`, renames the variable v into v' when it is used in guard of transitions after state s of STG \mathcal{S} . The second one is used for updating the set of free variables of each state in order to satisfy the STG definition (Definition 1).

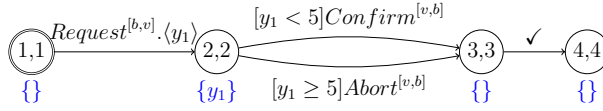


(a) Online-shopping choreography



(b) Implementation of Buyer

(c) Implementation of Vendor



(d) Composition of (b) and (c)

Figure 3.4: STGs for Example 1 and Example 2

Algorithm 1: Product of n STGs

Data: n STGs $\mathcal{S}_i = (S_i, s_{0i}, T_i)$
Result: a STG $\mathcal{S} = (S, s_0, T)$, the product of $\mathcal{S}_1, \dots, \mathcal{S}_n$

```

1   $s_0 := \{s_{01}, s_{02}, \dots, s_{0n}\}$ ; // initial state
2   $S := \{s_0\}$ ; // set of nodes
3   $T := \emptyset$ ; // set of transitions
4  product  $\{s_{01}, s_{02}, \dots, s_{0n}\}$ ;
5  // update set of free variables of each state to satisfy STG definition
6  updateFreeVariables  $(S, s_0, T)$ ;
7  return  $(S, s_0, T)$ ;
8
9  product  $\{s_1, s_2, \dots, s_n\} = \mathbf{begin}$ 
10 |  $s := \{s_1, s_2, \dots, s_n\}$ ;
11 | foreach  $s_1 \xrightarrow{[\phi_{i11}] \checkmark / s_{i11}}, s_2 \xrightarrow{[\phi_{i22}] \checkmark / s_{i22}}, \dots, s_n \xrightarrow{[\phi_{inn}] \checkmark / s_{inn}},$  do
12 |    $s' := \{s_{i11}, \dots, s_{inn}\}; \quad S := S \sqcup \{s'\}; \quad T := T \cup s \xrightarrow{[\phi_{i11} \wedge \dots \wedge \phi_{inn}] \checkmark / s'};$ 
13 |   foreach  $s_l \xrightarrow{[\phi_{il}] c^{[a,b]}!x_{il}/s_{il}}, s_k \xrightarrow{[\phi_{jk}] c^{[a,b]}?(x_{jk})/s_{jk}}$  with  $l \neq k \wedge l, k \in \{1, \dots, n\}$  do
14 |      $s'$  is constructed from  $s$  by replacing  $x_l$  by  $x_{il}$  and  $x_k$  by  $x_{jk}$ ;
15 |      $S := S \sqcup \{s'\}; \quad T := T \cup s \xrightarrow{[\phi_{il} \wedge \phi_{jk}] c^{[a,b]}.x_{jk}/s'};$ 
16 |     renameVariable  $(\mathcal{S}_l, s_l, x_{il}, x_{jk})$ ;
17 |     product  $(s')$ ;
18 |
19 | end

```

3.2.4 Reachability

STG is a specific directed graph where each transition is guarded by a condition. A transition t is never fired when its guard is always false for any value of variables, *e.g.*, $(x > 2) \wedge (x < 0)$. In such a case, the transition t is *unreachable*, otherwise it is reachable. If a transition t is unreachable then the transitions, which are only visited through t , are also unreachable. A STG is *reachable* if all its transitions are reachable. It is straightforward to see that the reachable STG obtained from a STG by removing all unreachable transitions has the same behaviors with the original one.

All unreachable transitions of a STG are cut off thanks to the pseudo-code in Algorithm 2. Particularly, given a STG (S, s_0, T) , we start the traversal from the initial state s_0 . For each outgoing transition t , we verify the combination of its guard and its precedent guards accumulating from the s_0 . If the combination is always *false*, then the transition is removed. Furthermore, all transitions which are only visited through t are also removed.

Algorithm 2: Reachable STG

```

Input: a STG  $\mathcal{S} = (S, s_0, T)$ 
Output: reachable STG
1 visit  $(\phi, s) = \text{begin}$ 
2   if  $(s \text{ is visited})$  then return;
3   foreach transition  $t_i = s \xrightarrow{[\phi_i] \alpha_i} s_i$  in  $\text{outgoing}(s)$  do
4     if  $\neg(\text{SOLVE}(\phi \wedge \phi_i))$  then
5       remove  $t_i$  from  $\mathcal{S}$  ; // cutoff
6       remove the transitions which depend only on  $t_i$  from  $\mathcal{S}$  ;
7     visit  $(\phi \wedge \phi_i, s_i)$ ;
8 visit  $(\text{true}, s_0)$  ;

```

In the sequel of this chapter, we examine only reachable STGs, *e.g.*, if a STG is not reachable, we must firstly removed all unreachable transitions.

3.3 Symbolic Conformance

In this section, we present our conformance relation between a model denoting the semantics of a set of local descriptions for interacting entities, \mathcal{I} , and a choreography global specification, \mathcal{C} . In the sequel, \mathcal{I} will be named *implementation* even if we have seen before that it may denote either a real implementation or a set of local requirements to be implemented. Since the semantic models are STGs, we define conformance over two STGs, one for \mathcal{I} and one for \mathcal{C} . We choose branching bisimulation [Van Glabbeek and Weijland, 1996] as a basis since it supports equivalence in presence of τ actions that result from the hiding of interactions added in imple-

mentations wrt. specifications, *i.e.*, refinement. However, branching bisimulation is defined over ground terms (no variables), while STGs may contain free variables.

In [Busi et al., 2006, Kazhamiakin and Pistore, 2006b], this issue is considered by introducing at each state an evaluation function that maps variables to values, thus reducing open terms to ground ones. Conformance is then verified only for some fixed values of the model variables. It is possible to check conformance for different fixed values. However, this may lead to state space explosion when domains of the variables are very big. As an alternative, we base our work on (late) symbolic extensions of bisimulations, introduced in [Hennessy and Lin, 1995, Lin, 1996, Li and Chen, 1999], that directly support open terms.

3.3.1 Making Implementation and Specification Comparable

First of all, we remind the reader that we assume the two STGs have disjoint sets of variables which can be achieved using, *e.g.*, indexing. We also assume for simplicity that a local entity has the same identifier than the corresponding role in the choreography. This constraint could be lifted using a mapping function. Additional interactions may have been introduced in the implementation *wrt.* the specification during refinement, *e.g.*, to make it realizable. In order to compare the STGs, we have first to hide these interactions, which is done using restriction (Definition 2).

Definition 2 (Restriction). *Given an STG $\mathcal{S} = (S, s_0, T)$ and a finite set of events \mathcal{E} , the restriction of \mathcal{S} to \mathcal{E} , denoted by $\mathcal{S} \downarrow_{\mathcal{E}}$, is the STG, \mathcal{S}' , obtained from \mathcal{S} by renaming into τ all the events that do not exist in \mathcal{E} and updating the set of free variables of each state in \mathcal{S}' in order to satisfy Definition 1.*

Example 4. *We give refinement examples in Figure 3.5 in which some interactions are introduced. In the first case (a), we have an additional interaction to agree on the phone (there is no choice) that acceptance will be used. In the second case (b), the client can specify the maximum (s)he agrees to pay for the articles (y). This influences the sequel of the implementation since the costs \$10: if the user requires to pay less, no shipping is done. The restriction of STGs in Figure 3.5 to the set of events used in the choreography specification, $\{\text{Request}^{[c,s]}, \text{Abort}^{[s,c]}, \text{Confirm}^{[s,c]}\}$, yields the STGs in Figure 3.6.*

3.3.2 Conformance Relation

Thanks to the previous steps (product of the local entity STGs and STG restriction), in the sequel we may consider STGs with interaction, termination, and hidden actions only, *i.e.*, $\hat{\alpha} \in \{o^{[a,b]}.x, o^{[a,b]}. \langle x \rangle, \checkmark, \tau\}$. Further, we know that the first of the two

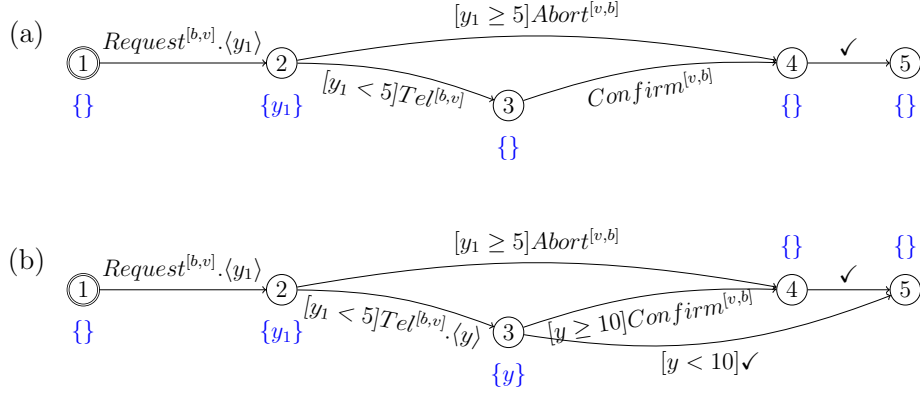


Figure 3.5: Two Refinements for the Implementation in Figure 3.4(d)

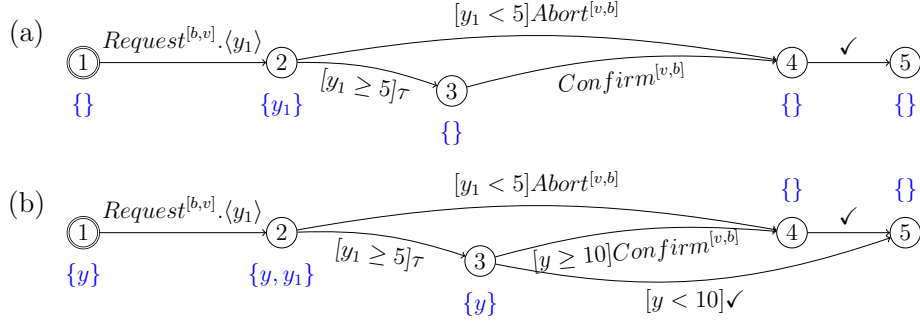


Figure 3.6: Restrictions of the STGs in Figure 3.5

STGs we compare may have τ events (resulting from restriction) while the second STG (the choreography specification) may not.

A key element is to be able to “absorb” series of τ s. The idea originates from weak forms of bisimulations such as weak bisimulation [Milner, 1999] and branching bisimulation [Van Glabbeek and Weijland, 1996] where we have a transition $s \xRightarrow{\tau} s'$ if we have a path of zero or more $\xrightarrow{\tau}$ transitions between s and s' . However, to work with STGs, these $s \xRightarrow{\tau} s'$ transitions have to be extended to support the guards that may appear on $\xrightarrow{\tau}$ transitions. This is achieved using rules EMPTY and TAU* in Figure 3.7. Rule SEM in this figure defines the semantics of $\xRightarrow{\tau}$ transitions. This semantics is denoted with relations $\xRightarrow{\tau}$ over terms in the same way than we had \rightarrow transitions in STG and \mapsto in their semantics. These rules are a simplification of the ones in [Li and Chen, 1999] that support $s \xRightarrow{\hat{\alpha}} s'$ transitions where $\hat{\alpha}$ is not always τ (this difference being a consequence of the definition of weak bisimulation wrt. branching bisimulation).

$$\begin{array}{ccc}
\text{(EMPTY)} & \text{(TAU*)} & \text{(SEMANTICS)} \\
\frac{}{s \xrightarrow{[true] \tau} s} & \frac{s \xrightarrow{[\phi_1] \tau} s_1, s_1 \xrightarrow{[\phi_2] \tau / s'} s'}{s \xrightarrow{[\phi_1 \wedge \phi_2] \tau} s'} & \frac{s \xrightarrow{[\phi] \tau} s'}{s_\sigma \xrightarrow{[\phi \sigma] \tau} s'_\sigma}
\end{array}$$

Figure 3.7: Retrieval of $\xRightarrow{\tau}$ Transitions and Their Semantics

In presence of variables and guards, the semantics of transition firing in STGs can be supported by domain enumeration as we have seen before. To avoid this (and the risk of state space explosion), we give a symbolic semantics to firing too, associating to transitions the condition under which it can be fired (Definition. 3).

Definition 3 (Fireable Transition). *A transition $t \xrightarrow{[\phi] \hat{\alpha}} t'$ (resp. $t \xRightarrow{[\phi] \tau} t'$) is fireable under condition ρ , $fv(\rho) \subseteq fv(\phi)$, iff $\rho \Rightarrow \phi$. In such a case, we write $\rho \models t \xrightarrow{[\phi] \hat{\alpha}} t'$ (resp. $\rho \models t \xRightarrow{[\phi] \tau} t'$). By extension, we write $\rho \models t \xRightarrow{[\phi] \tau} t' \xrightarrow{[\phi'] \hat{\alpha} / t''}$ if we have $\rho \models t \xRightarrow{[\phi] \tau} t'$ and $\rho \models t' \xrightarrow{[\phi'] \hat{\alpha}} t''$.*

Our conformance relation (Definition 4) is inspired by the extension of weak bisimulation into branching bisimulation [Van Glabbeek and Weijland, 1996] and the extension of weak bisimulation into symbolic weak bisimulation [Li and Chen, 1999]. We take into account termination (\checkmark) and the fact that there may be τ s only in the first of the two compared STGs.

Definition 4 (Symbolic Branching Bisimulation for Conformance - SBBC). *A binary relation over terms, \mathcal{R}^ρ , parametrized by a boolean formula ρ , is a symbolic branching bisimulation for conformance (SBBC) iff $(t_1, t_2) \in \mathcal{R}^\rho$ implies:*

1. $\forall (\rho \models t_1 \xrightarrow{[\phi_1] \tau} t'_1) \quad (t'_1, t_2) \in \mathcal{R}^\rho$
2. $\forall (\rho \models t_1 \xrightarrow{[\phi_1] \checkmark} t'_1) \quad \exists (\rho \models t_2 \xrightarrow{[\phi_2] \checkmark} t'_2)$
3. $\forall (\rho \models t_1 \xrightarrow{[\phi_1] c.\langle x_1 \rangle} t'_1) \quad \exists (\rho \models t_2 \xrightarrow{[\phi_2] c.\langle x_2 \rangle} t'_2) \quad (t'_1[x_1 \mapsto z], t'_2[x_2 \mapsto z]) \in \mathcal{R}^{\rho'}$
with $\rho' = \rho[x_1 \mapsto z, x_2 \mapsto z]$ and z is a fresh variable
4. $\forall (\rho \models t_1 \xrightarrow{[\phi_1] c.x_1} t'_1) \quad \exists (\rho \models t_2 \xrightarrow{[\phi_2] c.x_2} t'_2) \quad (t'_1, t'_2) \in \mathcal{R}^{\rho \cap (x_1 = x_2)}$
5. $\forall (\rho \models t_2 \xrightarrow{[\phi_2] \checkmark} t'_2) \quad \exists (\rho \models t_1 \xRightarrow{[\phi_1] \tau} t'_1 \xrightarrow{[\phi'_1] \checkmark} t''_1) \quad (t'_1, t_2) \in \mathcal{R}^\rho$
6. $\forall (\rho \models t_2 \xrightarrow{[\phi_2] c.\langle x_2 \rangle} t'_2) \quad \exists (\rho \models t_1 \xRightarrow{[\phi_1] \tau} t'_1 \xrightarrow{[\phi'_1] c.\langle x_1 \rangle} t''_1) \quad (t'_1, t_2) \in \mathcal{R}^\rho \wedge$
 $(t''_1[x_1 \mapsto z], t'_2[x_2 \mapsto z]) \in \mathcal{R}^{\rho'}$ with $\rho' = \rho[x_1 \mapsto z, x_2 \mapsto z]$, and z is a fresh variable

$$7. \forall (\rho \models t_2 \xrightarrow{[\phi_2] \text{ c.x}_2} t'_2) \quad \exists (\rho \models t_1 \xrightarrow{[\phi_1] \tau} t'_1 \xrightarrow{[\phi'_1] \text{ c.x}_1} t''_1) \quad (t'_1, t_2) \in \mathcal{R}^\rho \wedge (t''_1, t'_2) \in \mathcal{R}^{\rho' \cap (x_1=x_2)}$$

Definition 4 gives the conditions under which two terms, t_1 and t_2 , are R^ρ equivalent. Case (6) states that for an interaction fireable under condition ρ from t_2 to t'_2 , there must be an equivalent interaction fireable under condition ρ from t_1 to t''_1 , possibly after zero or more τ transitions between t_1 and some t'_1 . Equivalence of interactions is up to renaming of the used variables (x_1 and x_2) into a fresh variable z . Moreover, following branching bisimulation, we must have t'_1 and t_2 (respectively t''_1 and t'_2) equivalent. In the later case, we have to take into account the renaming of x_1 and x_2 by z in terms (t''_1 and t'_2) and in the equivalence relation condition (ρ). Case (5) is simpler. To a termination in t_2 corresponds a termination in t_1 reachable after zero or more τ s. Since no data is bound by termination, we just have to take into account recurrence over R^ρ . Cases (2), (3) and (4) are symmetric versions of cases (5), (6) and (7) respectively, simpler since there are no τ s in the specification/ t_2 . Finally, case (1) states that nothing is to correspond in t_2 to a τ transition in t_1 , but for the recurrence over R^ρ .

Given two STGs $\mathcal{I} = (S_1, s_{01}, T_1)$ and $\mathcal{C} = (S_2, s_{02}, T_2)$, we write $\mathcal{I} \simeq^\rho \mathcal{C}$ if there exists a SBBC relation \mathcal{R}^ρ between the terms formed by the two initial states with empty substitutions, *i.e.*, $(s_{01\emptyset}, s_{02\emptyset}) \in \mathcal{R}^\rho$. We can now give the formal definition of choreography conformance.

Definition 5 (Choreography Conformance). *Let C be a choreography specification and I be an implementation consisting of n local entities P_1, \dots, P_n , let \mathcal{C} be the STG for C , \mathcal{I} be the STG generated by the product of the STGs for P_1, \dots, P_n , and E be the alphabet of \mathcal{C} , I conforms to C iff $\mathcal{I} \downarrow_E \simeq^{\text{true}} \mathcal{C}$.*

3.3.3 Conformance Computation

Our algorithm for the computation of the SBBC relation between two STGs is a modification and simplification of the one proposed in [Li and Chen, 1999] that computes symbolic weak bisimulation. Simplification was made possible due to the use of SBBC for choreography conformance: there may be τ s in \mathcal{I} but not in \mathcal{C} . The algorithm outputs a set of boolean formulas ρ_{s_1, s_2} relative to pairs of states (s_1, s_2) , s_1 being in \mathcal{I} and s_2 in \mathcal{C} . ρ_{s_1, s_2} denotes the conditions under which s_1 and s_2 are SBBC related (Def. 4). In the algorithm, these boolean formulas are encoded as a Predicate Equation Systems (PESs) [Lin, 1996], *i.e.*, a set of predicate equations. A *predicate equation* (PE) is a function which contains a boolean expression, *e.g.*, $R(x) ::= (x \geq 0)$.

Following [Lin, 1996], a PES can be written using substitutions in order to simplify its notation. For example, the PES $\{R_0() ::= \forall z \ R_1(z, z); R_1(x, y) ::= (x > y)\}$ is rewritten $\{R_0 ::= \forall z \ R_1([x/z, y/z]); R_1 ::= (x > y)\}$.

The main function is function `close`, called initially on the initial states of the two STGs we compare. This function compares two states, s_1 and s_2 up to some substitution σ , and returns the most general formula such that s_1 and s_2 are SBBC bisimilar. This is formula *false* in the worst case (s_1 and s_2 are not branching bisimilar). `close` relies on the `match` function that encodes the different cases in Definition 4. $Trans(s_1, s_2) = \{\alpha | s_1 \xrightarrow{[\phi_1] \tau} s'_1 \xrightarrow{[\phi'_1] \alpha / s'_1} \} \cap \{\alpha | s_2 \xrightarrow{[\phi_2] \alpha / s'_2} \}$ denotes the common observable actions that can be performed both from s_1 and from s_2 . Since function `close` visits a pair $(s_1, s_2) \in S_1 \times S_2$ at most once, it always terminates after a finite steps, as for the algorithm whose complexity is $\mathcal{O}(|S_1| \times |S_2|)$.

Example 5. Applying the algorithm on the STGs in Figure 3.6(b) (i.e., restriction of Fig. 3.5(b)) and in Figure 3.4(a) (specification), we retrieve the following PES:

$$\begin{aligned}
R_{1,1}() &::= \forall Z_0 \ R_{2,2}(Z_0, Z_0) \\
R_{2,2}(y_1, x_1) &::= (((x_1 \geq 5 \Rightarrow y_1 \geq 5 \wedge R_{3,2}(y_1, x_1)) \wedge (y_1 \geq 5 \Rightarrow x_1 \geq 5 \wedge R_{3,2}(y_1, x_1))) \\
&\quad \wedge ((x_1 < 5 \Rightarrow y_1 < 5 \wedge R_{4,3}) \wedge (y_1 < 5 \Rightarrow x_1 < 5 \wedge R_{4,3}))) \\
&\quad \wedge (\neg(y < 10)) \\
R_{3,2}(y_1, x_1) &::= ((x_1 \geq 5 \Rightarrow y \geq 10 \wedge R_{4,3}) \wedge (y \geq 10 \Rightarrow x_1 \geq 5 \wedge R_{4,3})) \\
&\quad \wedge ((\neg(y < 10)) \wedge (\neg(x_1 < 5))) \\
R_{4,3}() &::= true
\end{aligned}$$

It can be simplified into $\{R_{1,1} ::= y \geq 10, R_{2,2} ::= y \geq 10, R_{3,2} ::= y \geq 10 \wedge Z_0 \geq 5, R_{4,3} ::= true, \}$ but this demonstrates the need for an automatic PES satisfiability checking procedure.

3.3.4 PES Satisfiability and Conformance Verdict

The formula resulting from conformance checking is under the form of a PES (see Section 3.3.3). It has to be analyzed in order to reach a conformance verdict. This step is performed with Z3, a state-of-the art theorem prover from Microsoft Research that can be used to check for the satisfiability of a set of formulas, i.e., find if there is an interpretation that makes all asserted formulas true.

In order to use Z3, we translate the PES into the Z3 input language as demonstrated in Listing 3.1 for the PES in Example 5. Each predicate equation in the PES is translated as a boolean function (using *define-fun*) and each free variable is translated as an integer function (using *declare-fun*). In our example, variables are integers, but we stress out that complex types such as XML ones can be supported too, following [Bentakouk et al., 2011].

Algorithm 3: Conformance PES Computation

Data: two STGs $\mathcal{I} = (S_1, s_{0,1}, T_1)$ and $\mathcal{C} = (S_2, s_{0,2}, T_2)$
Result: a PES, *i.e.*, a set P of PE's (one for each couple of states in a subset of $S_1 \times S_2$)

```

1  W :=  $\emptyset$  ;                                     // couples of visited nodes
2  P :=  $\emptyset$  ;                                     // predicates for couples of visited nodes
3  close( $s_{0,1}, s_{0,2}, \emptyset$ ) ;
4  return P ;

5  close( $s_1, s_2, \sigma$ ) = begin
6    if ( $s_1, s_2$ )  $\notin$  W then
7      W := W  $\cup$  {( $s_1, s_2$ )};     $R_{s_1, s_2} := \text{match}(s_1, s_2, \sigma)$ ;     $P := P \cup \{R_{s_1, s_2}\}$  ;
8    return  $R_{s_1, s_2}(\sigma)$  ;                                     // a predicate  $\rho_{s_1, s_2}$  with parameter  $\sigma$ 

9  match( $s_1, s_2, \sigma$ ) = begin
10   foreach  $\gamma \in \text{Trans}(s_1, s_2)$  do                               //  $\text{Trans}(s_1, s_2)$  is set of events  $\neq \tau$  of
11      $\rho_\gamma := \text{matchEv}(\gamma, s_1, s_2, \sigma)$  ;                 // next transitions from  $s_1, s_2$ 
12   // The others events ( $\notin \text{Trans}(s_1, s_2)$ ) must not occur
13   foreach  $s_1 \xrightarrow{[\phi'_{j,1}]\tau} s'_{j,1} \xrightarrow{[\phi_{j,1}]\alpha_{j,1}} s_{j,1}$  such that  $\alpha_j \notin \text{Trans}(s_1, s_2)$  do
14      $\rho_{j,1} := \neg\phi_{j,1}$  ;                                     // exist  $\tau$  in implementation
15   foreach  $s_2 \xrightarrow{[\phi_{i,2}]\alpha_{i,2}} s_{i,2}$  such that  $\alpha_i \notin \text{Trans}(s_1, s_2)$  do
16      $\rho_{i,2} := \neg\phi_{i,2}$  ;                                     // no  $\tau$  in specification
17   return  $\bigwedge_\gamma \rho_\gamma \wedge \bigwedge_j \rho_{j,1} \wedge \bigwedge_i \rho_{i,2}$  ;

18 // Success termination
19 matchEv( $\checkmark, s_1, s_2, \sigma$ ) = begin
20   foreach  $s_1 \xrightarrow{[\phi'_{j,1}]\tau} s'_{j,1} \xrightarrow{[\phi_{j,1}]\checkmark} s_{j,1}, s_2 \xrightarrow{[\phi''_{i,2}]\checkmark} s_{i,2}$  do
21     // if no  $\tau$  between  $s_1$  and  $s'_{j,1}$ , then  $s_{j,1}$  and  $s_{i,2}$  always conform
22     if  $s_1 \equiv s'_{j,1}$  then  $\rho_{ij} := \text{true}$ ; else  $\rho_{ij} := \text{close}(s'_{j,1}, s_2, \sigma)$  ;
23      $\phi_{j,1} := \phi'_{j,1} \wedge \phi''_{j,1}$  ;
24   return  $\bigwedge_i (\phi_{i,2} \Rightarrow \bigvee_j (\phi_{j,1} \wedge \rho_{ij})) \wedge \bigwedge_j (\phi_{j,1} \Rightarrow \bigvee_i (\phi_{i,2} \wedge \rho_{ij}))$  ;

25 // Bound interaction
26 matchEv( $c. <x>, s_1, s_2, \sigma$ ) = begin
27    $z := \text{newVar}()$  ;                                     // create a new fresh variable
28   foreach  $s_1 \xrightarrow{[\phi'_{j,1}]\tau} s'_{j,1} \xrightarrow{[\phi''_{j,1}]c.x_{j,1}} s_{j,1}, s_2 \xrightarrow{[\phi_{i,2}]c.x_{i,2}} s_{i,2}$  do
29     if  $s_1 \not\equiv s'_{j,1}$  then  $\phi_{j,1} := \phi'_{j,1}$ ;  $\phi_{i,2} := \text{true}$ ;  $\rho_{ij} := \text{close}(s'_{j,1}, s_2, \sigma)$ ;
30     else  $\phi_{j,1} := \phi''_{j,1}$ ;  $\rho_{ij} := \forall z \text{ close}(s_{j,1}, s_{i,2}, \sigma[x_{j,1} \mapsto z, x_{i,2} \mapsto z])$  ;
31   return  $\bigwedge_i (\phi_{i,2} \Rightarrow \bigvee_j (\phi_{j,1} \wedge \rho_{ij})) \wedge \bigwedge_j (\phi_{j,1} \Rightarrow \bigvee_i (\phi_{i,2} \wedge \rho_{ij}))$  ;

32 // Free interaction
33 matchEv( $c.x, s_1, s_2, \sigma$ ) = begin
34   foreach  $s_1 \xrightarrow{[\phi'_{j,1}]\tau} s'_{j,1} \xrightarrow{[\phi''_{j,1}]c.x_{j,1}} s_{j,1}, s_2 \xrightarrow{[\phi_{i,2}]c.x_{i,2}} s_{i,2}$  do
35     if  $s_1 \not\equiv s'_{j,1}$  then  $\phi_{j,1} := \phi'_{j,1}$ ;  $\phi_{i,2} := \text{true}$ ;  $\rho_{ij} := \text{close}(s'_{j,1}, s_2, \sigma)$ ;
36     else  $\phi_{j,1} := \phi''_{j,1}$ ;  $\rho_{ij} := (x_{j,1} = x_{i,2}) \wedge \text{close}(s_{j,1}, s_{i,2}, \sigma)$  ;
37   return  $\bigwedge_i (\phi_{i,2} \Rightarrow \bigvee_j (\phi_{j,1} \wedge \rho_{ij})) \wedge \bigwedge_j (\phi_{j,1} \Rightarrow \bigvee_i (\phi_{i,2} \wedge \rho_{ij}))$  ;

```

Listing 3.1: Translation into the Z3 Language of the PES in Example 5

```

1 ; encoding of the PES - it can be tried online at: http://rise4fun.com/
  z3
2 ; note: newlines should be added manually if copy/paste from PDF is used
3 (set-option :print-warning false)
4 (declare-fun y () Int)
5 (define-fun R4_3 () Bool true)
6 (define-fun R3_2 ((y_1 Int)(x_1 Int)) Bool (and (and (implies (>= x_1 5)
  (and (>= y 10) R4_3)) (implies (>= y 10) (and (>= x_1 5) R4_3))) (and
  (not (< y 10)) (not (< x_1 5)))))
7 (define-fun R2_2 ((y_1 Int)(x_1 Int)) Bool (and (and (and (implies (>=
  x_1 5) (and (>= y_1 5) (R3_2 y_1 x_1))) (implies (>= y_1 5) (and (>=
  x_1 5) (R3_2 y_1 x_1)))) (and (implies (< x_1 5) (and (< y_1 5) R4_3))
  (implies (< y_1 5) (and (< x_1 5) R4_3)))) (not (< y 10)))
8 (define-fun R1_1 () Bool (forall ((Z_0 Int)) (R2_2 Z_0 Z_0)))
9 ; uncomment for step 1, comment for step 2
10 (assert (= R1_1 false))
11 ; comment for step 1, uncomment for step 2
12 ; (assert (= R1_1 true))
13 (check-sat)

```

We check `R1_1` in order to conclude on conformance. For this, the `check-sat` Z3 command is run following Table 3.2. If `R1_1` asserted *false* (as in Listing 3.1) yields an *unsat* response then there is no interpretation such that $R_{1,1}$ is false, hence we can conclude directly that conformance is *true*. Otherwise, we have to retry with `R1_1` asserted to *true* to reach a verdict. If it responds an *unsat*, *i.e.*, there is no interpretation such that $R_{1,1}$ is true, then we can conclude that the conformance is *false*. If the response is *sat*, *i.e.*, there exists some interpretation such that $R_{1,1}$ is true; meanwhile there exists also some interpretation such that $R_{1,1}$ is false thanks to the former check. In such a case, we give the condition $\rho_{1,1}$ representing constraints of free variables such that the conformance can be achieved. Otherwise, the response is *timeout*, *i.e.*, Z3 can not solve the PES, hence we give *inconclusive* as the conclusion of conformance relation.

Table 3.2: Decision Table for Conformance based on PES Satisfiability Checking

check-sat response		conformance decision
R1_1 asserted <i>false</i>	R1_1 asserted <i>true</i>	
<i>unsat</i>	not needed	<i>true</i>
otherwise	<i>unsat</i> <i>sat</i> <i>timeout</i>	<i>false</i> $\rho_{1,1}(R_{1,1})$ <i>inconclusive</i>

3.4 Realizability Checking & Projection

A trivial implementation of a choreography is a single service which plays all roles of the choreography. In such a case, there is no more realizability issue. However, choreography intends to specify, from a global view, a collaboration of a set of roles. Each role in the choreography is a concrete entity taking part in this collaboration. It should be implemented by a distinguishable, independent service. Consequently, an *implementation* of a choreography should be a set of services where each one implements one of the roles and their composition represents the behaviors required by the choreography. This implementation can be constructed based on a projection and local conformance. The local conformance ensures that a service respects its role. It will be studied in the next chapter. This section is then dedicated to define the projection.

Example 6. *In order to illustrate the problems related to the realizability checking of service choreography, we extend the Online Shopping choreography presented in Example 1 as in Figure 3.8. We model the scenario using an extended BPMN 2.0 choreography [Knuplesch et al., 2012] as in Figure 3.8(a). It describes the collaborations between four independent participants: a buyer, a vendor, a warehouse, and a shipper. First, the buyer asks the vendor for an interested article by indicating the **name** of a requested article. The buyer responds with the information of the article **info**. This is repeated until the buyer decides to buy an article or it is aborted by the vendor. After receiving the buy request, the vendor issues a **sell** command to the warehouse. The warehouse replies with the status of the article. If it is available, the warehouse transfers the article to the shipper, then a confirmation will be issued from the vendor to the buyer. Otherwise, the vendor will notify a **sold** response to the buyer.*

*The STG corresponding to this choreography is presented in Figure 3.8(b). For simplicity, we denote b as the buyer, v as the vendor, w as the warehouse, and s as the shipper. The operation and variable names are also reduced. Since the information brought by **name** of the first interaction, **info**, **busy**, and **invoice** do not help any interactions, i.e., using by guard or free interaction, we remove them in the STG. The **name** in the third interaction is kept in the STG since it will be used by the **sell** and **ship** interactions.*

*Let us note that a specification specifies what rather than how system should or should not be done. Hence in this example, Figure 3.8, we do not pay attention to how the buyer selects the article itself, but after that we know the name of the selected one. The interaction **buy** should be described by a bound interaction, e.g., $\text{buy}^{[b,v]}. \langle x_1 \rangle$. Contrarily, in the interaction **sell**, the article name transferred from the vendor to the warehouse is the one the vendor received from the buyer, hence the*

interaction must be described by a free interaction, e.g., $sell^{[v,w]}.x_1$.

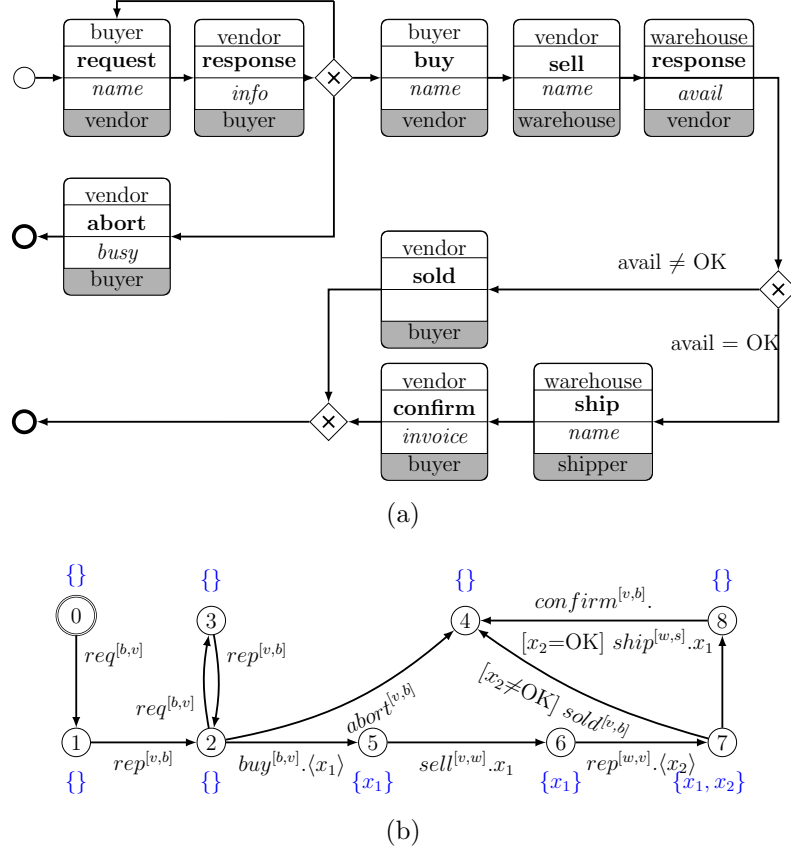


Figure 3.8: Online Shopping Process in (a) Extending BPMN 2.0 Choreography [Knuplesch et al., 2012] and in (b) Symbolic Transition Graph

The projection extracts relevant behaviors of role, local model, from a choreography. A set of such local models can be used as a candidate for implementation. Basically, *projection* is a procedure which takes as an input a choreography model with n roles and outputs a set of n local models, each one representing the required behaviors of a role in the choreography. Intuitively, behaviors of a local model are extracted from those of global model, in which a participates, e.g., the free interaction $o^{[a,b]}.x$ is projected onto free sending $o^{[a,b]}!x$ of role a , and on reception $o^{[a,b]}?x$ on role b .

Definition 6 (Natural Projection). *The natural projection of each transition of global STG is defined by rules in Table 3.3.*

Example 7 (Projection). *The projection of choreography STG in Figure 3.8(b) on its roles after removing two consecutive τ transitions is depicted in Figure 3.9.*

Table 3.3: Natural Projection of STG of Choreography

Transition	Project the transition		
	on role a	on role b	on role $c \notin \{a, b\}$
$s \xrightarrow{[\phi] o^{[a,b]}.x} s'$	$s \xrightarrow{[\phi] o^{[a,b]!}x} s'$	$s \xrightarrow{o^{[a,b]?}\langle x \rangle} s'$	$s \xrightarrow{\tau} s'$
$s \xrightarrow{[\phi] o^{[a,b]}. \langle x \rangle} s'$	$s \xrightarrow{[\phi] o^{[a,b]!} \langle x \rangle} s'$	$s \xrightarrow{o^{[a,b]?} \langle x \rangle} s'$	$s \xrightarrow{\tau} s'$

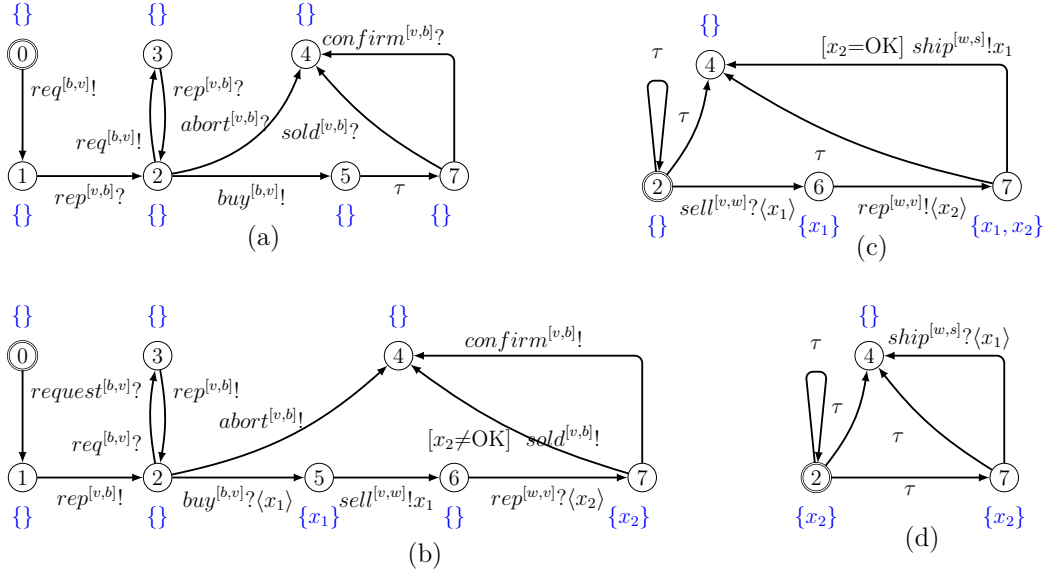


Figure 3.9: The Projection of STG in Figure 3.8(b) on Role: (a) buyer, (b) vendor, (c) warehouse, and (d) shipper

We now define the realizability issue by using the notions of projection as conformance as follow:

Definition 7 (Realizability). *A choreography is realizable iff there exists a (natural) projection function which generates a set of local models such that they (globally) conform to the choreography.*

Going further than determining whether a choreography is realizable, we intend to provide a dedicated projection which can be used also for unrealizable choreographies. In such a case, the projection is able to propose additional interactions in order to enable the realizability of a choreography. Our projection, which generates local models from a choreography, is performed in several steps. First, we remove all unreachable transitions of the choreography. We then work on a reachable choreography. We calculate a set of additional interactions needed to be added. If

this set is not empty, *i.e.*, the choreography is not realizable, then we minimize this set such that there are minimal interactions which are added. After adding the additional interactions to the choreography, we perform the natural projection of the choreography on each role. Finally, we reduce some consecutive transitions which contains only τ event. Let us go in detail by analyzing the cases where we need to introduce additional interactions. In the sequel of this section, we examine only reachable STGs, *e.g.*, if a STG is not reachable, we must firstly removed all unreachable transitions.

3.4.1 Event connectedness

On global models, an interaction, *e.g.*, α , which is a single event, becomes two separate events: a sending, $\alpha!$, and a reception, $\alpha?$, when it is projected on local models. The causality of the two local events depends on which communication model is considered. In synchronous communication mode, the sending and reception occur at the same time, denoted as $\alpha! = \alpha?$. In asynchronous one, the sending occurs before the reception, denoted as $\alpha! \prec \alpha?$.

The causality of two consecutive interactions, $\alpha_1 \prec \alpha_2$, is considered at local level by the causality of their sending and reception. In synchronous mode, we have $(\alpha_1! \prec \alpha_2!) \vee (\alpha_1? \prec \alpha_2?) \vee (\alpha_1! \prec \alpha_2?) \vee (\alpha_1? \prec \alpha_2!)$. In asynchronous mode, based on the results of [Lanese et al., 2008, Nguyen et al., 2012c], we consider three possibilities: $\alpha_1! \prec \alpha_2!$ (sending), $\alpha_1? \prec \alpha_2?$ (reception), and $\alpha_1? \prec \alpha_2!$ (disjoint). To ensure such causality of two consecutive interactions, α_1 and α_2 , the relations between their sender, s_1 and s_2 resp., and their receiver, r_1 and r_2 resp., must satisfy one of the conditions below, depending on the communication mode:

- synchronous mode: $\{s_1, r_1\} \cap \{s_2, r_2\} \neq \emptyset$
- sending mode: $s_2 = s_1 \vee s_2 = r_1$
- reception mode: $r_2 = r_1 \vee s_2 = r_1$
- disjoint mode: $s_2 = r_1$

These conditions enable all the participants to compute when they may or may not do (send or receive) the next interaction. Let us go into detail, for instance, with the sending mode assumption. This mode requires that the sending of the next interaction must happen after the one of the current interaction. If these two interactions have the same sender, *e.g.*, $s_2 = s_1$, then the sender always enables this condition. Moreover, if the sender of the next interaction and the receiver of the

current one are the same, *e.g.*, $s_2 = r_1$, when it receives the current interaction, it knows that the sending of the current interaction has already occurred, and thus it may enforce the condition by sending the second interaction.

If two consecutive transitions, *e.g.*, $s \xrightarrow{[\phi_1]\alpha_1} s_1 \xrightarrow{[\phi_2]\alpha_2} s_2$, with α_1 and α_2 do not satisfy the condition above, we introduce an additional transition at s_1 , *e.g.*, $s \xrightarrow{[\phi_1]\alpha_1} s' \xrightarrow{\alpha} s_1 \xrightarrow{[\phi_2]\alpha_2} s_2$. The event α does not carry a variable but it connects α_1 and α_2 . Sender and receiver of α is determined based on communication modes. If the sending mode is selected, then the sender of α is either the sender or the receiver of α_1 , while the receiver of α is either the sender α_2 . Thus with the additional interaction, the connectedness between α_1 and α , then between α and α_2 is guaranteed. For example, to ensure the connectedness of **ship** and **confirm** in our running example, with the sending mode, an additional interaction from the shipper to the buyer is added as follows: $s \xrightarrow{\text{ship}^{[w,s]}.x_0} s' \xrightarrow{\text{additional}^{[w,v]}} s_1 \xrightarrow{\text{confirm}^{[v,b]}.x_3} s_2$. Let us remark that, the interaction $\text{additional}^{[s,v]}$ may be also added.

3.4.2 Data connectedness

In a choreography specification, a variable indicates not necessarily explicitly its owned roles, *e.g.*, it is usually used to constrain data exchanges carried by interactions, since the specification represents *what* should be implemented rather than *how* can be implemented. However, in implementation, to validate a guard, a role must know the free variables that appear in the guard. Hence any free variables used by a role (in guards or in free interactions) must be known by itself. Intuitively, a variable is known when its value is pre-configured before running time or its value is received from another role. Let us remark that, value of variable x can be changed after a bound interaction $o^{[a,b]}.x$ occurs, hence only a and b know x when this interaction happens, other roles could know x if there exists some free interactions carrying x from either a or b to them.

If a role c uses a variable x in its guards or in its free interactions, but c does not know x , *e.g.*, $s \xrightarrow{o_1^{[a,b]}.x} s_1 \xrightarrow{[x>0] o_2^{[c,d]}.x} s_2$, then an additional interaction will be introduced in order to connect data of variable x from the roles knowing it, a or b , to c , *e.g.*, $s \xrightarrow{o_1^{[a,b]}.x} s' \xrightarrow{\text{additional}^{[b,c]}.x} s_1 \xrightarrow{[x>0] o_2^{[c,d]}.x} s_2$. The proposition of additional interactions in order to connect data is done as in Algorithm 4. In which a proposed additional interaction is in the form $\langle t, \{a, b\}, c, x \rangle$ where t is the transition which uses variable x , hence additional interaction will be inserted before it, and $\{a, b\}$ is a set of roles which knows x and will send x to c .

Algorithm 4: Correction of Data Connectedness

Input: a global STG $\mathcal{C} = (S, s_0, T)$
Output: set of additional interactions

```

1 let  $owner(s, x)$  as set of roles which know variable  $x$  before state  $s$ ;
2  $E := \emptyset$ ; // set of additional interactions
3 foreach transition  $s_i \xrightarrow{[\phi_i] \alpha_i} s'_i$  in  $T$  do
4    $V := \mathbf{fv}(\phi_i) \cup \mathbf{fv}(\alpha_i)$ ; // set variable to be verified
5    $a := \mathbf{sender}(\alpha_i)$ ;
6   foreach  $v \in V$  do
7     if ( $a \notin owner(s_i, v)$ ) then add  $\langle t, owner(s_i, v), a, v \rangle$  to  $E$ ;
8   // update  $owner$ 
9   if ( $\alpha_i$  is a bound interaction  $o^{[a,b]}. \langle x \rangle$ ) then  $owner(s'_i, x) := \{a, b\}$ ;
10  if ( $\alpha_i$  is a free interaction  $o^{[a,b]}.x$ ) then  $owner(s'_i, x) := owner(s_i, x) \cup \{b\}$ ;
11 return  $E$ ;
```

3.4.3 Branching decision

A state s of a STG may have many outgoing transitions. Each transition has a guard. If these guards are not implied each other, then the branching at the state s is firstly decided by these guards. Obviously, this condition cannot be determined at syntax level but at semantic level. In such a case, s is a *conditional branching*. Thanks to these guards, only one branch can be selected at a time, and each initial role can always compute whenever it does or does not initiate the interaction.

Contrarily, if the state s is an *unconditional branching*, there are several guards that may be satisfied, hence many transitions may be fired. The branching will be decided by the roles which initiate interactions of each outgoing transition. A transition $s \xrightarrow{[\phi] \alpha} s'$ may be initiated by either sender or receiver of α in synchronous mode, but only by the sender of α in asynchronous mode. State (2) of the STG in Figure 3.8(b) is an unconditional branching, while the state (7) is a conditional branching.

When many roles may decide the route, but these roles work independently, the choreography will not be respected since each role may select a different branch. Therefore, only one initiator is allowed to decide the route. If a branching has more than one initiator, a dominant initiator will be selected to decide the route. *Dominant initiator* is the role which may initiate maximal transitions, *e.g.*, at the state (2), *buyer* is a dominant initiator. Some additional interactions from the initiator to the other ones will be introduced in order to complete the control of the dominant initiator into these branches. Hence the selection of dominant initiator adds minimally interactions. Some additional interactions from the initiator to the other roles may be also introduced to inform them of the selected route.

As the natural projection, see Table 3.3, guards are preserved only at local model of the sender. However, in the case of a conditional branching, guards are also maintained at

local models which decide the route based on these guards. A conditional branching can be also decided as done for an unconditional branching. However, the conditional branching (based on data) is *better* than unconditional one (based on events) since less additional interactions are added. Let us consider a state s having m branches, of a choreography STG with n roles. If s is a conditional branching, then we need no more than n additional interactions which correspond to the n roles involved in the branching. Otherwise, if s is an unconditional branching, there are no more than $(n - 1) \times m$ additional interactions from dominant initiator to $n - 1$ other roles in order to inform selected branches. Most of existing works do not deal with the case of conditional branching, *i.e.*, all branchings are considered as unconditional ones, hence they usually insert more additional interactions than we do. The correction of branching is done as depicted by the pseudo-code in Algorithm 5.

Algorithm 5: Correction of Branching

```

Input: a choreography  $\mathcal{C} = (S, s_0, T)$ 
Output: set of additional interaction
1  let  $\mathcal{R}$  as set of roles in  $\mathcal{C}$ ;
2   $E := \emptyset$ ; // set of additional interactions
3  foreach  $s \in S$  do
4      if ( $|\text{outgoing}(s)| = 1$ ) then continue; // no branching --> pass to the next iteration
5       $\text{isConditional} := \text{true}$ ;
6      foreach transition  $s \xrightarrow{[\phi_1] \alpha_1} s_1, s \xrightarrow{[\phi_2] \alpha_2} s_2$  in  $\text{outgoing}(s)$  do
7          // whether exists some value of variables s.t.  $\phi_1$  and  $\phi_2$  are satisfied
8          if ( $\text{SOLVE}(\phi_1 \wedge \phi_2)$ ) then
9               $\text{isConditional} := \text{false}$ ; break;
10     if ( $\text{isConditional}$ ) then continue; // conditional branching
11     // unconditional branching
12     select a dominant initiator  $a$  that participates the most of outgoing transition;
13     foreach transition  $t_i := s \xrightarrow{[\phi_i] \alpha_i} s_i$  in  $\text{outgoing}(s)$  do
14          $b := \text{sender}(\alpha_i)$ ;  $c := \text{receiver}(\alpha_i)$ ;
15         foreach  $r \in \mathcal{R}$  do
16             if ( $(a \neq r) \vee (a \neq b \wedge r \neq c)$ ) then add  $\langle t, \{a\}, r \rangle$  to  $E$ ;
17 return  $E$ ;

```

The output of the projection of the global STG, described in Figure 3.8(b) under synchronous communication mode, produces four local STGs, as shown in Figure 3.10, corresponding to local models of the buyer (a), the vendor (b), the warehouse (c), and the shipper (d). For the sake of clarity, in these local STGs, the additional states are gray and the additional interactions start with +, *e.g.*, $+br_i$.

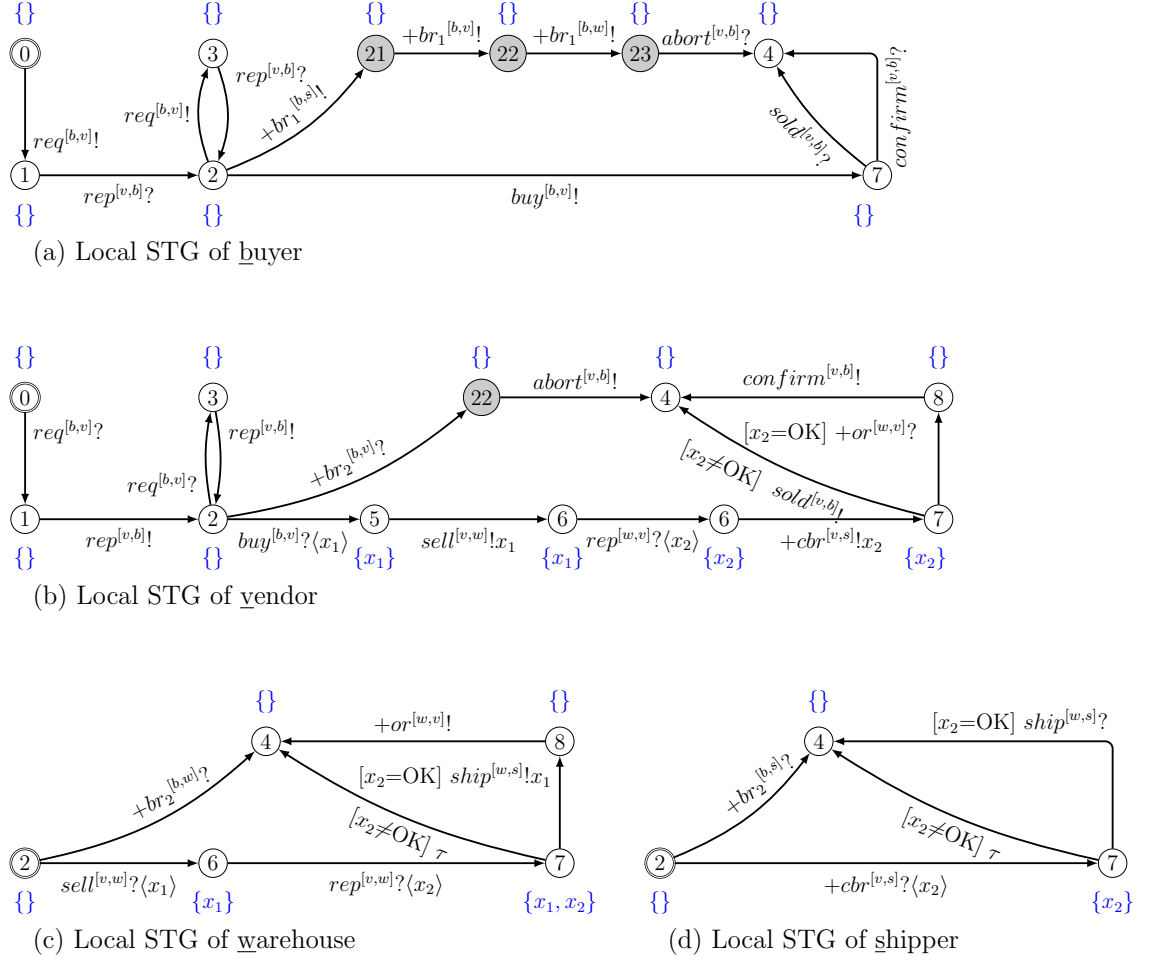


Figure 3.10: Projection with Additional Interactions of STG in Figure 3.8(b) under Synchronization Communication Mode

3.5 Tool & Experimental Evaluations

In this section, we now give in detail our tool-chain for analyzing service choreographies. Our tool-chain may be used directly from our Web pages¹. It can be also downloaded to use at local computer. The Z3 SMT solver² has to be installed separately for licence reasons. We plan to interface our tool-chain with the SMT-LIB API in order to let users choose other SMT solvers. We also present some of the experiments we have made to evaluate it.

¹<http://schora.lri.fr>

²<http://research.microsoft.com/en-us/um/redmond/projects/z3/>

3.5.1 Boolean Condition Solver

The unreachable transition and conditional branching, *e.g.*, the function `SOLVE(ϕ)` in the above algorithm, are checked with the aid of the Z3 SMT solver. Let us consider an example of checking whether the branch at state 7 in Figure 3.8(b) is a conditional branching. We need to check whether there exists some values of x_2 such that both the first branching condition ($x_2 = \text{OK}$) and the second branching condition ($x_2 \neq \text{OK}$) are *true*. The translation in a Z3 script to check this example is as in Listing 3.2.

Specially, we first declared a data sort `A` as a general data type. Then `OK` (resp. x_2) is declared as a constant (resp. variable) typed `A`. Two boolean functions ϕ_1 and ϕ_2 are defined based on relation between x_2 and `OK`. This very simple example shows that these functions, variable and constant are uninterpreted, *i.e.*, Z3 does not use their interpretation (concrete values) but relies on their definition supported by a dedicated decision procedure. The `check-sat` command checks if equation $(\phi_1 \wedge \phi_2)$ defined by the `assert` command is *true* for some interpretations of its variables. If this is the case, the response of `check-sat` is `sat`, otherwise, `unsat`, *i.e.*, the equation is always *false*. In the example, the response will be *unsat*.

Listing 3.2: Example of Condition Solver using Z3 SMT

```

1 (declare-sort A)
2 (declare-const OK A)
3 (declare-fun x2 () A)
4 (define-fun phi1 () Bool (= x2 OK))
5 (define-fun phi2 () Bool (not (= x2 OK)))
6 (assert (and phi1 phi2))
7 (check-sat)

```

3.5.2 Tool Architecture

Conformance Checking. The architecture of our tool-chain for conformance checking is given in Figure 3.11. It takes as input a choreography global specification C , with m roles. It also takes an implementation description I , given as $n \geq m$ entity local descriptions. These may correspond either to service descriptions or to role requirements. The case when $n > m$ denotes, *e.g.*, an implementation where some services have been added to make a choreography realizable. All inputs are first transformed into STGs. The product of STGs and the restriction to actions in C are used to retrieve a unique STG for I , thus yielding two STGs to compare: one for C (\mathcal{C}) and one for I (\mathcal{I}). We then check if \mathcal{I} conforms to \mathcal{C} , which generates the largest boolean formula ρ such that the initial states of \mathcal{I} and \mathcal{C} are SBBC related. Finally, this formula is analyzed using the Z3 SMT solver in order to reach a conformance verdict. This can be “always true” or “always false”, “always” meaning whatever the data values exchanged between services are. However, sometimes we

can have conformance only for a subset of these values. Going further than pure true/false conformance, our tool-chain thus allows to compute the largest constraint on data values, ρ , that would yield conformance. Complex constraints may cause the solver to return a timeout. In such a case, we emit inconclusiveness as a verdict.

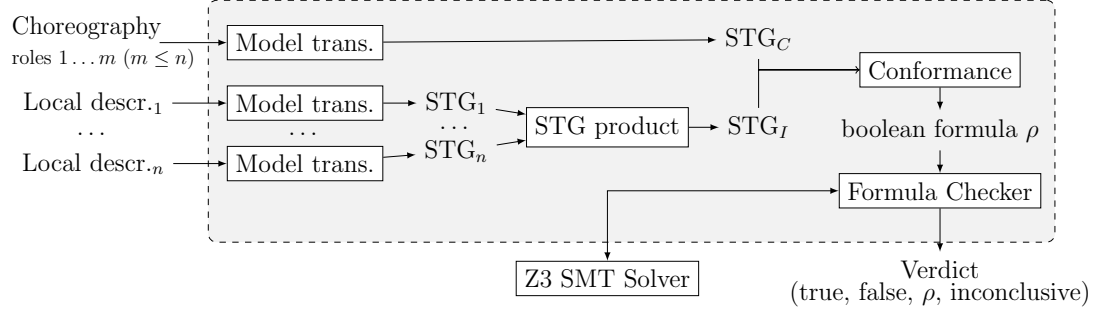


Figure 3.11: Architecture of our Toolchain for Conformance Checking

The tool is extended such that it can take directly STGs as inputs, hence there is no need of transformation. Moreover, one can also compose local services arbitrarily, then provide the tool with the STG product.

Realizability Checking. The architecture of the tool-chain for realizability checking and projection of service choreographies is presented in Figure 3.12. It takes as inputs two parameters, a choreography global specification with m roles and a communication mode which can be synchronous (SYNC), sending (ASYNC_SENDER), reception (ASYNC_RECEIVER) or disjoint (ASYNC_DISJOINT). The choreography specification is then translated into STG. The reachability checking will remove all unreachable states and transitions of the STG. If the reachable STG above is unrealizable, the realizability checking add minimally interactions to the STG to make it become realizable. The natural projection is then applied on the realizable STG.

3.5.3 Experimental Evaluation

Evaluation on Existing Case Study. We have experimented our tool-chain, including on examples from the literature: Market [Busi et al., 2006], Request For Quota (RFQ [Kazhamiakin and Pistore, 2006b]), and Train Station Services (TSS [Salaün et al., 2012]).

The experimental results of conformance checking is presented in Table 3.4. For the implementations and the specifications, we respectively give the numbers of operations (#Ops.), transitions (#Trans.), states (#States), services (#Services) and roles (#Roles). We also give the conformance verdicts in the paper the example

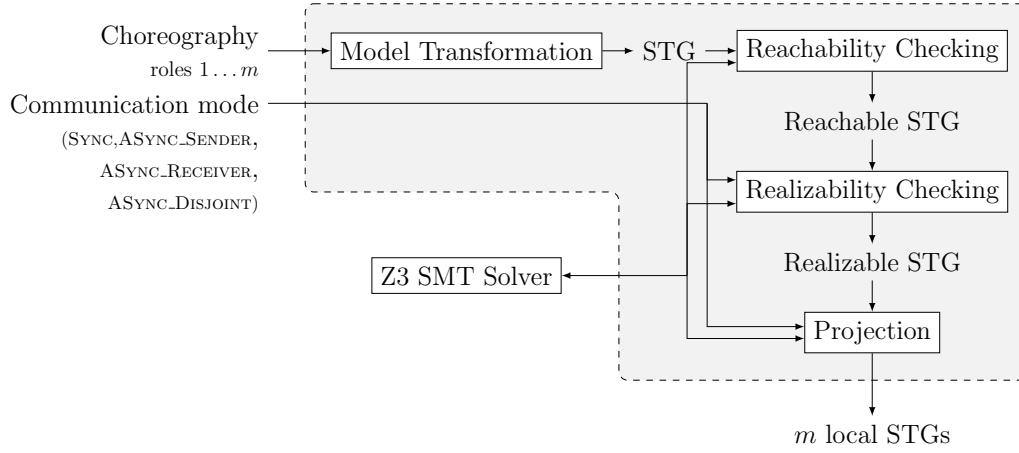


Figure 3.12: Architecture of our Toolchain for Choreography Projection

is taken from and with our approach. Finally, we give the execution time (Mac Book Air with OS 10.7, 4 GB RAM, core i5 1.7 GHz) for the process described in Figure 3.11 (but for the time to parse the input files). Rows 1 to 3 correspond to the specification STG in Figure 3.4(a) and, respectively, to the implementations STGs in Figures 3.4(d) (row 1), 3.5(a) (row 2), and 3.5(b) (row 3). Rows 4 and 5 correspond to the example and mutation in [Busi et al., 2006]. The difference in the verdict comes from the fact the we distinguish between an STG ending with \checkmark (successful termination) or not, hence an implementation deadlocking after achieving all interactions of a specification will not conform to it: the specification may do \checkmark while the implementation may not. Row 6 corresponds to a negative example in [Kazhamiakin and Pistore, 2006b] and row 7 to a positive one in [Salaün et al., 2012].

Table 3.4: Experimental Results of Conformance Checking

Name	Implementation (#Ops./ #Trans./States/#Services)	Specification (#Ops./ #Trans./States/#Roles)	Verdict (Orig./Ours)	Duration (seconds)
Example 1	3/4/4/2	3/4/4/2	-/YES	0.069
Mutation	4/5/5/2	3/4/4/2	-/YES	0.084
Example 5	4/6/5/2	3/4/4/2	-/ ρ	0.102
Market	8/9/10/2	8/10/10/4	YES/NO	0.118
	16/27/26/8	8/10/10/4	YES/NO	0.201
RFQ	6/8/7/3	6/8/8/3	NO/NO	0.078
TSS	8/12/11/4	8/12/11/4	YES/YES	0.096

The experimental results of realizability checking is shown in Table 3.5. It was done on a desktop computer running 32-bit XUbuntu 13.4 (kernel 3.8.0-23-generic) with Intel Pentium 4 3.2GHz processors and 1GB of RAM. The second column corresponds to

the inputs which are choreography specifications. The remaining columns are devoted to represent the checking result of: the reachability with their verdicts (Vdict) and number of cut-off interactions (#delInt.); the realizability, with their verdicts (Vdict) and number of additional interactions (#addInt.), corresponding to the four cases of communication modes, synchronous (sync.), sending, reception, and disjoint. The last column gives the time needed for the checking. To verify our projection, on one hand, we recomposed the generated local models to obtain one STG, representing the composition, which was then compared against the choreography STG. On the other hand, since local models of the examples were available, we compared them with our generated local models. The comparisons were done thanks to the conformance checking.

Table 3.5: Experimental Results of Projection & Realizability Checking

Name	Choreography Size (#Ops./#Trans./#States/#Roles)	Reachability (Vdict./#delInt.)	Realizability (Vdict./#addInt.)				Duration (seconds)
			sync.	sending	reception	disjoint	
Example 6	11/9/4/9	yes/0	no/5	no/7	no/7	no/7	0.060
Market	8/10/10/4	yes/0	yes/0	yes/0	no/1	no/1	0.077
RFQ	6/8/8/3	yes/0	yes/0	yes/0	no/1	no/1	0.041
TSS	8/12/11/4	yes/0	yes/0	no/2	no/1	no/4	0.125

Generic Experiments. Additional interactions are generated to solve the conflicts related to choreography order interaction, to branches selection and to variables values. The Figure 3.13 illustrates how the number of additional interactions is related to the number of states, branches for each state, roles, operations and data for the choreography under the assumption of synchronous communication mode. To conduct the experiments, we started from an initial configuration defined by 5 parameters and their values, *i.e.*, 1000 states, 5 branches, 5 roles, 20 operations, 1 data level. With this configuration we generated a choreography STG with a tree structure and it is a 5-ary tree. Data level 1 denotes that variables are carried by events for all the transitions of the choreography. Data level $n > 1$ means that variables are still carried by events and are also on guards. Consequently, the data level increases with the number of variables in the guard. Based on the initial configuration, we produced several other configurations by only varying one parameter. For instance, to test the impact of the number of states, as illustrated in Figure 3.13(a), we used 10 configurations: (states: x , branches: 5, roles: 5, operations: 20, data: 1) with $x \in \{1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000\}$. Based on each configuration, we generated randomly 30 choreography STGs. We then tested on each generated STG. The result of one tested configuration is the average of the results of these 30 STGs.

In our formal model, only one variable is changed when an interaction happens. Hence if a role on the next transition depends on this variable, we need to introduce

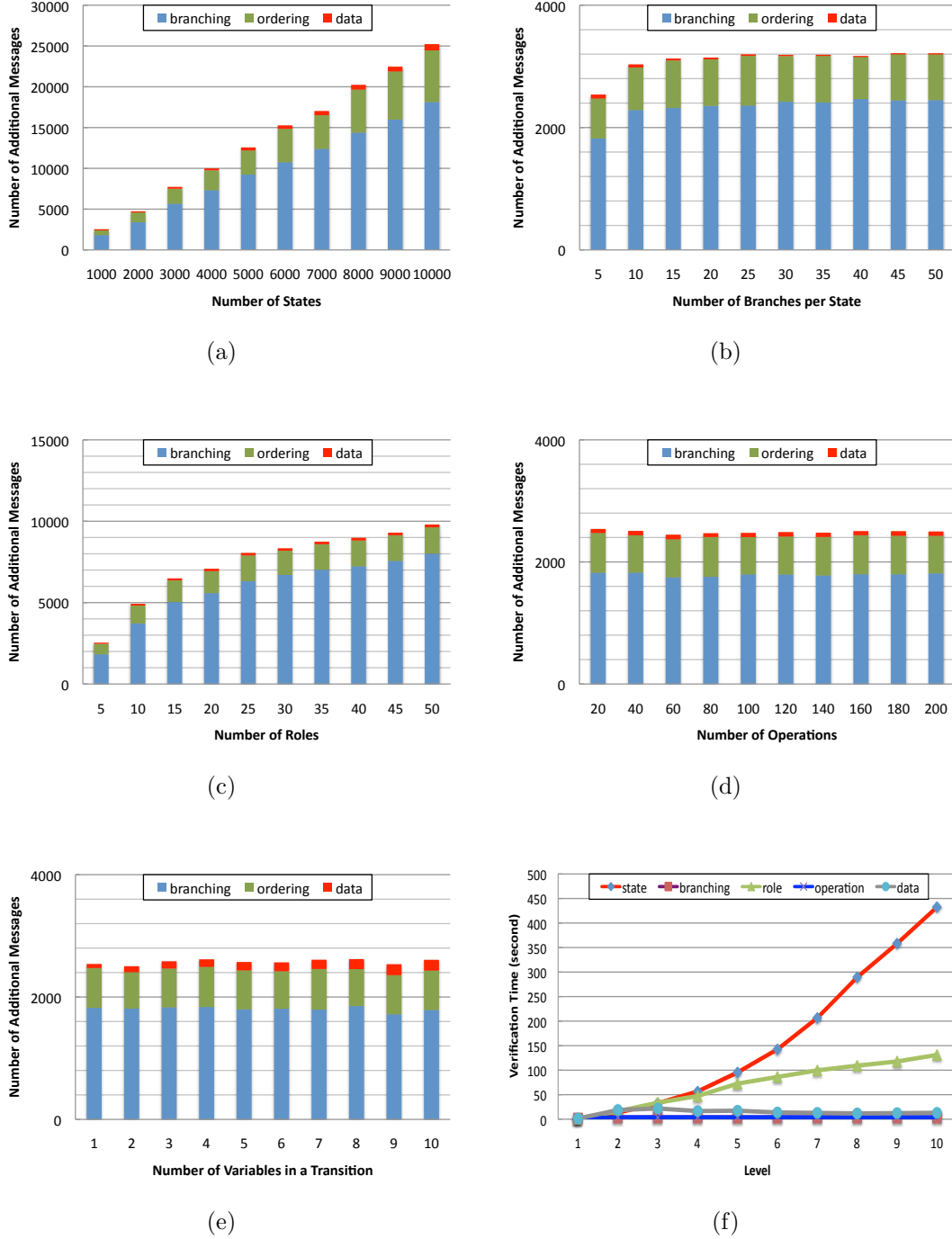


Figure 3.13: Impact of Number of (a) States, (b) Branches, (c) Roles, (d) Operations and (e) Variables on the Realizability of Choreography; and (f) Verification Time

only one additional transition to transfer this variable to the concerned role. To ensure the ordering of interactions, additional events are added. Indeed, when an interaction e occurs, m interactions may happen after e , *e.g.*, the target state of e may have m outgoing transitions. In such a case, there are no more than m additional interactions to add in order to solve the order conflict between e and m interactions. For the branching decision, as explained in Section 3.4.3, there are no more than $(n - 1) \times m$ additional interactions from a dominant role. This latter has to inform $n - 1$ roles of the selected branches. The number of additional interactions for each configuration consists of three parts, for data, branching and ordering. Generally, the proportions of these parts, data:ordering:branching, are $1:m:m \times (n - 1)$.

To confirm our analysis on the formal model, we have conducted several experiments that are depicted in Figure 3.13. The mentioned proportions are confirmed by the obtained results for each bar of the bar chart. Furthermore, we can exhibit with these experiments several features of our projection. In Figure 3.13(a), we observe that the number of interactions increases with the number of states. In this case, the tree becomes deeper. In Figure 3.13(b), when we vary the number of branches, we obtain a larger tree with a higher number of leaves, as a consequence the number of intermediate states will decrease. This decrease will slow down the increase of the number of additional interactions used for branching decisions. However the number of intermediate states decreases slowly when number of branches reaches the value 25 and higher values. That is why we observe that the number of additional interactions reaches a threshold. Let us note that for each experiment only one parameter was changed and especially here the number of states was not changed, it is why we obtain a larger tree. Consequently, for projection concerns, it is notably better to construct a choreography in breadth rather in depth. Figure 3.13(c) shows that our method is more efficient with a higher number of roles. Indeed, in this case, for each state, we determine a dominant role which will have in charge to inform the others of the selected route. However the dominant role does not need to inform a role which is a receiver of an interaction e on a branch since it is informed by the sender of e . Consequently, the dominant role needs to inform less roles when the number of roles (act as receivers) increases. Hence this slows down the increase of the number of additional interactions by increasing the number of roles. In Figure 3.13(d), we increase the number of operations, in others words the alphabet. Such a change has no impact on the choreography realizability as the complexity of the choreography (initial configuration) is not modified. Indeed, only the label, *i.e.*, operations name, are changed. In Figure 3.13(e) the number of variables is changed. We observe that when we increase the number of variables, the number of additional interactions increased in order to ensure the data-connectedness (the top bar is increasing). To conclude, hopefully the complexity of the projection with value-passing is not totally dependent on the data complexity. The verification times are shown in Figure 3.13(f).

The horizontal axis represents the 10 configurations. It illustrates once again the dependability of realizability on the complexity of choreography.

3.6 Discussion

Data may yield over-approximation when abstracted and/or state space explosion problem. Moreover, the presence of data in model produces a new attribute of model called *reachability* which intends to verify that all states of the model are reachable from the initial one. Thus, the unreachable states and transitions of a choreography model may be removed from the model. Consequently, this may change the realizability of choreography, *e.g.*, the unrealizability is caused by some unreachable events. The presence of data in model also produces a new verdict, called **Maybe**, of conformance checking. This verdict is emitted when the conformance is only achieved with some sub data domains of variables, *i.e.*, there is no conformance when variables receive value outside these domains.

This chapter introduced our symbolic framework which allows to model and analyze service choreographies in presence of data without suffering from state space explosion and without bounding data domains. Based on our model, we analyzed three fundamental issues of service choreography, *e.g.*, conformance checking, realizability checking, and projection. Our framework was fully implemented by our online tool which is a web application, *i.e.*, one can use it directly from any Web browser without any installation.

Having both loops and assignments may yield state space explosion if one does not close the system or bound data domains. In this work, we support loops and support a limited form of assignment through message reception. The assignment will be fully supported in the next chapter by using Symbolic Transition Graph with Assignments. However to avoid state space explosion, we must limit with a depth k the analysis of choreography, *e.g.*, the unfolding or computation of traces.

Passive Testing of Choreographies

Contents

4.1	Passive Conformance Testing	68
4.2	Online Property-Oriented Testing	81
4.3	Discussion	97

When models of choreography implementation are available, the correctness of the implementation can be verified by conformance checking presented in Chapter 3. When they are not available, testing is an alternative. This chapter is dedicated to introduce our two approaches of passive testing service choreographies. Section 4.1 presents our first approach which was also done firstly in thesis. This work applies naïve passive testing to test conformance of service choreographies in which choreography is specified by *Chor* language [Qiu et al., 2007]. In this work, we present also, in detail, our infrastructure used for passive testing of service choreographies. It also takes into account different scenarios of message correlations. After proposing our symbolic model described in Chapter 3, we studied the second approach of service choreography testing, presented in Section 4.2, which is based on property-oriented passive testing. This work has overcome limitations of the first one, *e.g.*, online testing, value-passing, and without requirement of global clock to synthesize global log. We then present our tools to validate the proposed approaches. Finally, Section 4.3 discusses our approaches and points out future work.

4.1 Passive Conformance Testing

4.1.1 *Chor* Language & Trace Semantics

A model with trace semantics is suitable for model-based testing. We started our work to test choreographies by using the *Chor* [Qiu et al., 2007] language for the specification of choreographies. While being simpler than more general purpose languages such as BPMN or UML, *Chor* is expressive and abstract enough to enable one to specify collaborations. *Chor* can also be seen as an abstraction of the WS-CDL [World Wide Web Consortium, 2005] standard. *Chor* defines both a choreography language, a role language, and projections between global and local (role) descriptions supporting the formal treatment of collaborations at both viewpoints.

The *Chor* Language and Its Semantics. The syntax of a *Chor* choreography specification is defined in Figure 4.1(a) for structuring activities (A) and for basic activities (BA). *skip* denotes a do-nothing action, while $c^{[i,j]}$ represents a basic interaction on some medium c between two roles of the choreography, namely i and j , called performers of the interaction. Since *Chor* is concerned about the abstract specification of the collaboration, the instantiation of some interaction medium, and details associated to it (e.g., exchanged data) is part of the developer duties. This is can be, e.g., using message and message parts in a Web service framework. With reference to [Qiu et al., 2007], we restrict to the observable fragment of *Chor*, i.e., we do not take into account the specification at the global level of local non-observable actions. Structuring in *Chor* is achieved using sequencing (;), exclusive choice ($+i$) and parallel flows ($|$), in which “;” has higher priority while “+” and “|” have the same priority.

$A ::= BA$ (basic activities) $ A; A$ (sequential) $ A + A$ (choice) $ A A$ (parallel) $BA ::= skip$ (no action) $ c^{[i,j]}$ (communication)	Basic: $\llbracket skip \rrbracket \triangleq \{\langle \rangle\}$ $\llbracket c^{[i,j]} \rrbracket \triangleq \{\langle c^{[i,j]} \rangle\}$ Sequential: $\llbracket A_1; A_2 \rrbracket \triangleq \llbracket A_1 \rrbracket \frown \llbracket A_2 \rrbracket$ Choice: $\llbracket A_1 + A_2 \rrbracket \triangleq \llbracket A_1 \rrbracket \cup \llbracket A_2 \rrbracket$ Parallel: $\llbracket A_1 A_2 \rrbracket \triangleq \llbracket A_1 \rrbracket \bowtie \llbracket A_2 \rrbracket$

(a) Syntax
(b) Semantics

Figure 4.1: *Chor* Choreography Language

The semantics of a *Chor* specification C is given in terms of its trace set, $\llbracket C \rrbracket$, i.e., the set of all possible traces of its execution, where a trace is a sequence $[\alpha_1, \dots, \alpha_n]$ of interactions. In the sequel we use t, t_1 , etc. for traces, T, T_1 , etc. for trace sets, and \llbracket

denotes the empty trace set. The semantics of *Chor* is given in Figure 4.1(b). It relies on concatenation (\wedge) and interleaving (\bowtie) operators defined in Figure 4.2. Further, we have $head([\alpha_1, \alpha_2, \dots, \alpha_n]) = [\alpha_1]$ and $tail([\alpha_1, \alpha_2, \dots, \alpha_n]) = [\alpha_2, \dots, \alpha_n]$.

Concatenation	Interleaving
$t \wedge T \triangleq \{t \wedge t_1 \mid t_1 \in T\}$ $T \wedge t \triangleq \{t_1 \wedge t \mid t_1 \in T\}$ $T_1 \wedge T_2 \triangleq \{t_1 \wedge t_2 \mid t_1 \in T_1, t_2 \in T_2\}$	$t_1 \bowtie t_2 \triangleq \begin{cases} \{t_1\} & \text{if } t_2 = [] \\ \{t_2\} & \text{if } t_1 = [] \\ head(t_1) \wedge (tail(t_1) \bowtie t_2) \cup head(t_2) \wedge (t_1 \bowtie tail(t_2)) & \text{otherwise} \end{cases}$ $T_1 \bowtie T_2 \triangleq \{t \mid \exists t_1 \in T_1 \wedge t_2 \in T_2 \text{ such that } t \in t_1 \bowtie t_2\}$

Figure 4.2: Operators on Traces

The *Chor* Role Language and its Semantics. Role requirements are described in a dialect of *Chor* called role languages (*Role* for short) with the only difference that a global interaction $c^{[i,j]}$ corresponds in role i (resp. j) to an emission denoted $c^{[i,j]}$! (resp. reception denoted $c^{[i,j]}$?). The semantics of a *Chor* (local or global) specification C is given in terms of its set of all specification traces, *trace set* for short, that represent all possible run of the specification [Qiu et al., 2007]. In the sequel, \wedge denotes trace concatenation and \bowtie denotes trace interleaving. Further in a trace, \boxtimes denotes a deadlock (*i.e.*, a blocking termination).

Getting Roles from Choreography. The requirements for each role of a choreography can be obtained using natural projection (*nproj*) hiding the interactions that do not concern the role of interest and orienting the other ones; *i.e.*, for a role k , the projection of $c^{[i,j]}$ gives $c^{[i,j]}$! if $k = i$, $c^{[i,j]}$? if $k = j$, and *skip* otherwise.

Example 8. Let us take the example of a collaboration involving two roles, a buyer (r_1) and a vendor (r_2). The buyer first issues a request to the vendor (c_1) which then gives back good information (c_2) and information of some concerned accessories (c_3). This is modelled in *Chor* as the specification $C_1 = c_1^{[1,2]}; (c_2^{[2,1]} | c_3^{[2,1]})$, whose semantics, $\llbracket C_1 \rrbracket$, is:

$$\begin{aligned} \llbracket C_1 \rrbracket &= \llbracket c_1^{[1,2]} \rrbracket \wedge (\llbracket c_2^{[2,1]} \rrbracket \bowtie \llbracket c_3^{[2,1]} \rrbracket) = \{\langle c_1^{[1,2]} \rangle\} \wedge \{\langle c_2^{[2,1]}, c_3^{[2,1]} \rangle, \langle c_3^{[2,1]}, c_2^{[2,1]} \rangle\} \\ &= \{\langle c_1^{[1,2]}, c_2^{[2,1]}, c_3^{[2,1]} \rangle, \langle c_1^{[1,2]}, c_3^{[2,1]}, c_2^{[2,1]} \rangle\} \end{aligned}$$

Using *nproj*, we may obtain a process for each role in C_1 :

$$\begin{aligned} r_1 &= nproj(C_1, 1) = c_1^{[1,2]}!; (c_2^{[2,1]}? | c_3^{[2,1]}?) \\ r_2 &= nproj(C_1, 2) = c_1^{[1,2]}?; (c_2^{[2,1]}! | c_3^{[2,1]}!) \end{aligned}$$

$ \begin{array}{lcl} P & ::= & BP \quad (\text{basics}) \\ & & P; P \quad (\text{sequential}) \\ & & P \sqcap P \quad (\text{choice}) \\ & & P P \quad (\text{parallel}) \\ BP & ::= & skip \quad (\text{no action}) \\ & & c^{[i,j]}! \quad (\text{send message}) \\ & & c^{[i,j]}? \quad (\text{receive message}) \end{array} $	$ \begin{array}{lcl} \text{Basic:} & skip \xrightarrow{\parallel} \varepsilon & \text{Local:} \quad a \xrightarrow{[a]} \varepsilon \\ \text{Sequential:} & \frac{P_1 \xrightarrow{\sigma} P'_1}{P_1; P_2 \xrightarrow{\sigma} P'_1; P_2} & \varepsilon; P_2 \xrightarrow{\parallel} P_2 \\ \text{Choice:} & P_1 + P_2 \xrightarrow{\parallel} P_1 & P_1 + P_2 \xrightarrow{\parallel} P_2 \\ \text{Parallel:} & \varepsilon \varepsilon \xrightarrow{\parallel} \varepsilon & \\ & \frac{P_1 \xrightarrow{\sigma} P'_1}{P_1 P_2 \xrightarrow{\sigma} P'_1 P_2} & \frac{c? \in fst(P_1) \quad c! \in fst(P_2)}{P_1 P_2 \xrightarrow{[c]} P_1 / c? P_2 / c!} \\ & \frac{P_2 \xrightarrow{\sigma} P'_2}{P_1 P_2 \xrightarrow{\sigma} P_1 P'_2} & \frac{c! \in fst(P_1) \quad c? \in fst(P_2)}{P_1 P_2 \xrightarrow{[c]} P_1 / c! P_2 / c?} \end{array} $	$ \begin{array}{lcl} fst(\varepsilon) = fst(skip) \triangleq \emptyset \\ fst(P_1 + P_2) \triangleq \emptyset \\ fst(\alpha) \triangleq \{\alpha\} \\ fst(P_1; P_2) \triangleq fst(P_1) \\ fst(P_1 P_2) \triangleq fst(P_1) \cup fst(P_2) \\ skip/\alpha \triangleq \perp \\ \alpha/\alpha' \triangleq \begin{cases} \varepsilon & \text{when } \alpha = \alpha' \\ \perp & \text{when } \alpha \neq \alpha' \end{cases} \\ (P_1; P_2)/\alpha \triangleq P_1/\alpha; P_2 \\ (P_1 + P_2)/\alpha \triangleq \perp \\ (P_1 P_2)/\alpha \triangleq \begin{cases} P_1/\alpha P_2 & \text{when } \alpha \in fst(P_1) \\ P_1 P_2/\alpha & \text{when } \alpha \in fst(P_2) \\ \perp & \text{otherwise} \end{cases} \\ \frac{P \xrightarrow{\sigma} P'}{P \xrightarrow{\sigma} P'} & \frac{P \xrightarrow{\sigma} P' \quad P' \xrightarrow{\sigma'} P''}{P \xrightarrow{\sigma \hat{\circ} \sigma'} P''} \end{array} $
(a) Syntax & Semantics	(b) Operators on trace	

Figure 4.3: *Role* language

$ \begin{aligned} nproj(c^{[i,j]}, k) &\triangleq \begin{cases} c^{[i,j]}! & \text{if } k = i \\ c^{[i,j]}? & \text{if } k = j \\ skip & \text{otherwise} \end{cases} \\ nproj(skip, k) &\triangleq skip \end{aligned} $	$ \begin{aligned} nproj(A_1; A_2, k) &\triangleq nproj(A_1, k); nproj(A_2, k) \\ nproj(A_1 A_2, k) &\triangleq nproj(A_1, k) nproj(A_2, k) \\ nproj(A_1 + A_2, k) &\triangleq nproj(A_1, k) + nproj(A_2, k) \end{aligned} $
--	---

Figure 4.4: Natural Projection of *Chor* Language

The trace sets of r_1 , r_2 , and $r = r_1 | r_2$, that represent respectively all possible executions of r_1 , r_2 , and their collaboration:

$$\begin{aligned}
\llbracket r_1 \rrbracket &= \{ \langle c_1^{[1,2]}!, c_2^{[2,1]}?, c_3^{[2,1]}? \rangle, \langle c_1^{[1,2]}!, c_3^{[2,1]}?, c_2^{[2,1]}? \rangle \} \\
\llbracket r_2 \rrbracket &= \{ \langle c_1^{[1,2]}?, c_2^{[2,1]}!, c_3^{[2,1]}! \rangle, \langle c_1^{[1,2]}?, c_3^{[2,1]}!, c_2^{[2,1]}! \rangle \} \\
\llbracket r_1 | r_2 \rrbracket &= \{ \langle c_1^{[1,2]}, c_2^{[2,1]}, c_3^{[2,1]} \rangle, \langle c_1^{[1,2]}, c_3^{[2,1]}, c_2^{[2,1]} \rangle \}
\end{aligned}$$

4.1.2 Local & Global Conformance

Formal methods provide many techniques to verify conformance between a specification and an implementation, *e.g.*, using behavioral equivalences and preorders [Bergstra et al., 2001]. This has been applied recently to component and service based architectures [ter Beek et al., 2007]. However, in our context, we have an important constraint: the implementation source code is un-available since it is made up of

distributed black-box services. Some approaches suppose that such services have behavioral interfaces, but in practice this is seldom the case. A recent proposal enables to retrieve such interfaces from black-box services using testing [Bertolino et al., 2009]. Still, this can be an intrusive technique, *i.e.*, that cause change on the service due to the active nature of the test being used. In our context we base on non intrusive passive testing techniques. Implementations may only be observed and checked for conformance using their (execution) logs, which are (linear) sequences of observations. This advocates for the use of a trace equivalence or a trace preorder.

Further, the relation between a choreography specification C and an implementation I can be seen with two mirror perspectives. In the former perspective, it is *the coordination middleware* that is tested. We are interested in the fact that I strictly enforces (over connected services) what is described in C . I should then exhibit at least (or exactly) the behavior described in C . This may be supported using active testing of service orchestrations [Bozkurt et al., 2010]. In the second perspective, it is *the cooperation of the services* that is tested. We are interested in the fact that the services do not interact in some other ways than what is specified in C . This corresponds to a passive testing using logs at the services' locations. In such a case, we impose that the traces of I are included in the C ones. In this work, we focus on the second perspective. We may now give our formal definition of the conformance of an implementation with reference to a specification. For this we base on trace preorder.

Definition 8 (Preorder). *The preorder relation, denoted with \preceq , between two traces is defined recursively as follows:*

- $\langle \rangle \preceq \langle \sigma_2 \rangle$; and
- $\langle \alpha, \sigma_1 \rangle \preceq \langle \alpha, \sigma_2 \rangle$ iff $\langle \sigma_1 \rangle \preceq \langle \sigma_2 \rangle$.

Given two trace sets T_1 and T_2 , we write $T_1 \preceq T_2$ iff $\forall t_1 \in T_1, \exists t_2 \in T_2 \mid t_1 \preceq t_2$.

Indeed, while implementing a choreography, the developer may have to add important additional exchanges or synchronizing activities in the services. This is especially the case with non-realizable choreographies. Take for example the specification $C = c_1^{[1,2]}; c_2^{[3,4]}$. Projecting it on its roles we get $R_1 = c_1^{[1,2]}!$, $R_2 = c_1^{[1,2]}?$, $R_3 = c_2^{[3,4]}!$, and $R_4 = c_2^{[3,4]}?$. Implementing the specification using these four services as-is, *i.e.*, $I = R_1|R_2|R_3|R_4$, the developer cannot prevent that R_3 and R_4 interact on c_2 before R_1 has sent c_1 to R_2 : trace $[c_2^{[3,4]}, c_1^{[1,2]}]$ is in $\llbracket I \rrbracket$ while it is not in $\llbracket C \rrbracket = \{\langle c_1^{[1,2]}, c_2^{[3,4]} \rangle\}$. Therefore, the developer may decide to add a synchronizing message between R_2 and R_3 , c_{sync} , to enforce the choreography, *i.e.*, replacing R_2 and R_3 above respectively

by $c_1^{[1,2]?}; c_{\text{sync}}^{[2,3]}!$ and $c_{\text{sync}}^{[2,3]?}; c_2^{[3,4]}!$. However, in such a case, we would not have conformance, *i.e.*, $\llbracket I \rrbracket \not\preceq \llbracket C \rrbracket$. To support this, we formally define conformance as follows.

Definition 9 (Preorder Trace Equivalence). *Given a specification S and an implementation I . We have $I \text{ conf } S$ iff $\llbracket I \rrbracket \downarrow_{\text{acts}(S)} \preceq \llbracket S \rrbracket$, where $\text{acts}(S)$ is the set of all activities of S and \downarrow is the filter operator, *i.e.*, $t \downarrow_X$ (or $T \downarrow_X$) retains only the elements of X in t (or T) while preserving their order.*

Before going on, let us stress a basic assumption that we make on the relation between a choreography, C , and an implementation of it, I . We suppose a one-to-one function between the roles in C and the services in I : each role r_i is implemented by exactly one service s_i , and each service s_i implements exactly one role r_i . This enables us to relate service communications in log files to role activities in specifications. Based on the formal definition of conformance we proposed above, we may now define conformance between a choreography implementation and a choreography specification.

Definition 10 (Choreography Conformance). *Given a choreography C with n roles, $r_i = \text{nproj}(C, i)$, and an implementation I with n services, s_i , I conforms to C , denoted $I \text{ conf } C$, iff the following two conditions hold:*

- *local conformance: $s_i \text{ conf } r_i$, for every $i = 1..n$, and*
- *global conformance: $(S_1|S_2|\dots|S_n) \text{ conf } C$.*

Example 9. *To illustrate the need of both local and global conformances, let us take again the C_1 choreography specifications from Example 8, and examine the conformance of an implementation I_1 made up of two services, S_1 and S_2 , whose logs are $t_1 = [c_1^{[1,2]}!; c_2^{[2,1]}?]$ and $t_2 = [c_1^{[1,2]}?; c_2^{[2,1]}!; c_2^{[2,1]}!]$ respectively. S_2 obviously does not conform to r_2 since in the later only one $c_2^{[2,1]}!$ may happen, while two ones appear in the log. S_1 has realized $c_1^{[1,2]}!; c_2^{[2,1]}?$, *i.e.*, an unique reception of $c_2^{[2,1]}?$. The second message $c_2^{[2,1]}!$ sent by S_2 may have been lost or cancelled. Hence S_1 conforms to r_1 . In the model composition $S = S_1|S_2$, only the $c_1^{[1,2]}$ and $c_2^{[2,1]}$ interactions have been done. While S conforms to C_1 (Definition 9), I_1 does not conform to C_1 since S_2 does not conform to R_2 (Definition 10).*

4.1.3 Implementation

In *active testing*, the tester interacts with the IUT by sending inputs (messages) and observing outputs (messages too). This method assumes a kind of controllability of

the implementation through Points of Control and Observations (PCOs). Observing the outputs and comparing them to the expected ones, *i.e.*, those described by the specification, a verdict can be emitted. A **Pass** verdict establishes the conformance of the implementation to its specification and a **Fail** the contrary. *Passive testing* is a software testing method that relies only on observations on the running IUT. In passive testing the tester does not send messages to the IUT. It only observes the exchange (sending and reception) of messages between the IUT and its partners, through POs. These observations will be compared to the specification in order to emit a verdict.

In both cases, active and passive testing, the implementation is considered as a black box, which means that the internal structure of the implementation is not known and no source code is available. The term “passive” relates to the fact that the tests do not disturb the natural operation of the IUT, to the contrary of “active” testing. Passive testing is also of particular interest since we do not always have the ability to control an IUT. Using passive testing in our work, testing can be done continuously and the services in a collaboration can evolve dynamically. Such a seamless monitoring activity is a less costly activity as it does not require to make the IUT unavailable during the testing process.

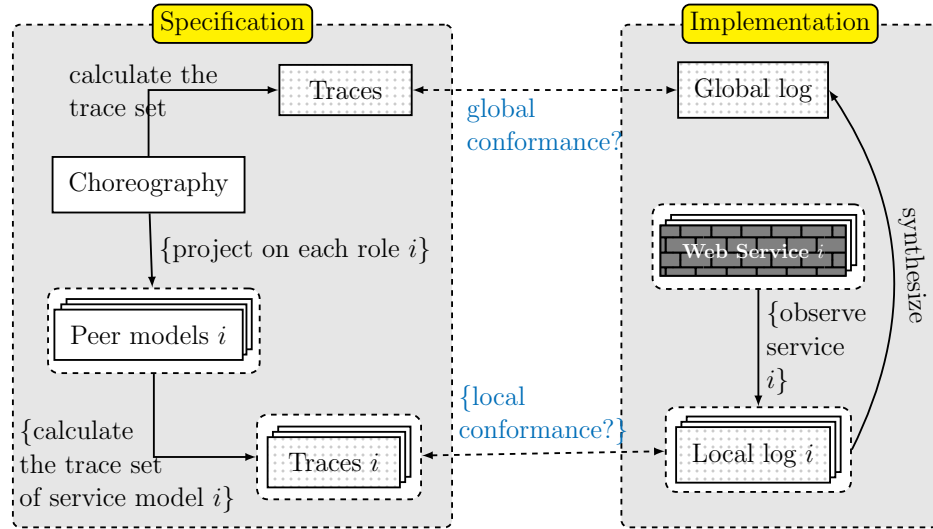


Figure 4.5: Conformance Testing Choreography Implementations

Our testing approach is represented by Figure 4.5. In this figure, repeated activities are denoted with the “{” and “}” symbols, *e.g.*, $\{Project\ on\ each\ role\ i\}$ means that the projection will be repeated on each role r_i of C . At the specification level, from a choreography specification C , we can obtain directly its trace set ($\llbracket C \rrbracket$). Afterwards, the set of local requirements r_i of its roles is obtained by using the natural projection

function. From r_i we have also its trace set ($\llbracket r_i \rrbracket$). At the implementation level, we can obtain local implementation traces, called logs and denoted as l_i , from each PO. From n collected logs, we synthesize a global log, denoted as \overline{log} . Global log, \overline{log} , and local logs, l_i , will be checked against $\llbracket C \rrbracket$ and $\llbracket r_i \rrbracket$ to establish global conformance and local conformance, respectively. By Definition 10, the IUT conforms with C if and only if both global and local conformance are achieved.

Our approach is fully supported by a tool that realizes *Chor* specification parsing, trace semantics retrieval, global log synthesis, and conformance checking. We have also implemented a monitoring module for the Apache ODE BPEL engine in order to retrieve local logs.

4.1.4 Observation of SOAP Messages

The approaches to capture SOAP messages in the context of Web services (WSs) can be classified into three groups. The first approach injects some modules into the WS engine in order to extract the expected information, *e.g.*, [Wu et al., 2008, Moser et al., 2008a, Simmonds et al., 2009, Moser et al., 2008b]. The second approach intends to implement a software which performs as a SOAP “proxy” and which is independent and external to the WS engine, *e.g.*, SoapUI¹ or Membrane SOAP/HTTP Monitor². All incoming and outgoing SOAP messages of the WS must be directed to this SOAP proxy, which then forwards the messages to their destination while conserving a copy of them. The last approach sniffs passively the SOAP messages which are transferred in the network by means of sniffers such as tcpdump³ or wireshark⁴.

The two last approaches capture the SOAP messages when they are outside of a WS engine, and are as a consequence independent from it. This means that they can be used for different types of Web service engines. However these two approaches, and also more generally all approaches which only capture the SOAP messages outside WS engines, miss some important features which are necessary for testing. They cannot guarantee that the captured SOAP messages will be sent to the destination and that these messages are accepted by the Web service partner. Another problem is that they cannot know which instance of a Web service sends (or receives) the captured messages. Let us consider the following example. Web service r_1 can send its requests either to r_2 or r_3 . This is described as $r_1 = c_1^{[1,2]}! + c_2^{[1,3]}!$. This example raises the problem a “false negative” verdict in case we capture two consecutive messages c_1 and c_2 at the PO of r_1 . The verdict has to be a **Fail** if these messages

¹<http://www.soapui.org/>

²<http://www.membrane-soa.org/soap-monitor/>

³<http://www.tcpdump.org/>

⁴<http://www.wireshark.org/>

are sent by the same instance of the WS r_1 , but if they are sent by two different instances the verdict has to be **Pass**.

With its advantages we chose the first approach to implement our tool to collect the SOAP messages. The architecture of our monitor consists of a module integrated into Apache ODE, a WS-BPEL compliant WS orchestration engine. It allows to capture all messages which are sent or received by a monitored service and to record it into a log file. However as this module is integrated into the ODE engine, it may cause a limitation by considering only the monitoring of services which are implemented in WS-BPEL. To overcome this issue, we have also implemented a kind of wrapper which adds a WS-BPEL layer for services which are not WS-BPEL Web services. Hence, we are able to capture and to log all the SOAP messages that are transmitted between the monitored service and its partners.

Each service is observed by a monitor that captures all the input and output messages of the service. The captured messages that do not concern the testing choreography will be discarded by the tester. When a message is sent or received by a service of the IUT, an *observation* is recorded immediately in the log file. A *Chor* interaction $c^{[i,j]}$ means that a message c is transferred from r_i to r_j . As a consequence, an observation contains the sender, the receiver, and the type of the message. Moreover, at the specification (role) level, an interaction is equipped with “!” or “?” to indicate if it is a sending or a reception. Hence, we annotate observations accordingly in logs. To ease the reconstruction of the order between observations of different logs l_i , an observation also contains the time of the observation. Furthermore, to correlate the observations of a sending message and of the corresponding reception one, we inject, at the sending moment, an identity in the header of SOAP messages.

Formally, the observation is defined as follows. Given an IUT which consists of n services $S = \{S_1, \dots, S_i, \dots, S_n\}$, an *observation* is a tuple $ob = (act, id, t, s, r, m)$ where $act \in \{\text{SEND}, \text{RECEIVE}\}$ indicates a sending or a reception, id is a message identity, t is the reception or sending time, $s, r \in S$ with $s \neq r$ are the sender and the receiver, and m is the message. A log $l_i = [ob_1, ob_2, \dots, ob_m]$ for a service s_i is a sequence of all the observations of messages which are sent/received by s_i to/from others services of the IUT.

4.1.5 Global Log Synthesis

We have a set of local logs l_i and we need to synthesize a global log in order to perform global conformance testing, *i.e.*, compare this global log with the trace set of the choreography. As proposed in [Zaïdi et al., 2009], we assume that clocks of services are synchronous to support the construction of an order between two observations that have happened in two different local logs, *e.g.*, two . This assumption typically holds,

e.g., when collaborations are deployed over clouds. Communication is synchronous in *Chor*. This means that the sending and the reception events for an interaction happen at the same time. In an implementation there is usually a delay between a sending and the corresponding reception. A choreography $C = c_1^{[1,2]}; c_2^{[3,4]}$ means that the passing of message c_1 from r_1 to r_2 should happen before the passing of message c_2 from R_3 to R_4 . Figure 4.6 represents all possible correlations between the sending and the reception time of messages c_1 and c_2 , where the order of execution is denoted with an arrow “ \rightarrow ”. For example, in Figure 4.6(a), the sending of c_2 only happens after the reception of c_1 . One can note that the case represented in Figure 4.6(a) is the strongest one, *i.e.*, it implies the other cases. For example, if $c_1^{[1,2]?}$ happens before $c_2^{[3,4]!}$ as in Figure 4.6(a), we can infer that $c_1^{[1,2]!}$ happens before $c_2^{[3,4]!}$ as in Figure 4.6(c) because of execution order transitivity (*i.e.*, $a \rightarrow b \wedge b \rightarrow c \Rightarrow a \rightarrow c$). Hence case (a) is the default in our tool. However one may select any other case in the tool.

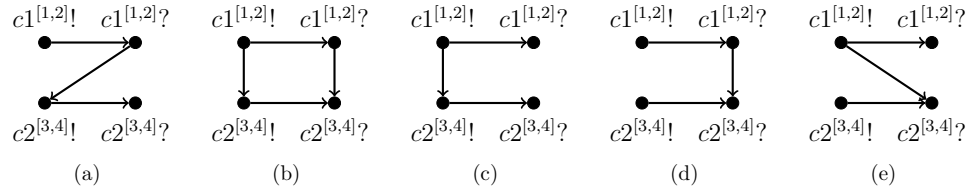


Figure 4.6: Sendings and Receptions Correlation

Algorithm 6 represents the synthesis of global log based on the first case. This algorithm is divided into two parts. The first part (lines 1–17) creates a new list of observations by merging n local logs. The observations in this list are sorted by the order of execution time. In fact, each local log has partial ordered, so that this part is basically to sort n arrays which are already sorted. The second part (lines 18–27) is to synthesize the global observations from the sorted list of local observations corresponding to the case represented in Figure 4.6(a), which imposes that, in logs, the reception of a message has to be adjacent to the sending of this message. This means that there is nothing (neither sending nor reception of other messages) which can happen in this interval.

4.1.6 Testing Algorithm

Before presenting the algorithms for global and local conformance verification, we present Algorithm 7 which verifies the preorder relation between a log and a trace. It is an implementation of Definition 8.

The algorithm for global conformance verification is presented in Algorithm 8. The implementation to realize the choreography may need additional interactions. In

this case, we only keep the interactions which are present in the specification by using the filtering function. As soon as we find a trace of the choreography such that the global log is its preorder then this log passes the conformance checking and the algorithm is stopped. If we visit the whole trace set without finding any trace of the specification such that the global log is its preorder, this means that the implementation is not exhibiting a behavior represented by the specification, which entails to return a **Fail**. The algorithm for local conformance verification is similar. The only difference is that we do not have to synthesize the global log from n local logs, *i.e.*, line 2 in Algorithm 8.

Algorithm 6: Synthesis of Global Observations ($synthesisObservations(L)$)

```

Input: A set of  $n$  logs  $\overline{log} = [log_1, log_2, \dots, log_n]$ 
Output: A log  $log = \{ob_1, ob_2, \dots, ob_m\}$ 
1  /* Get global order of observations */
2   $t = []$ ;
3   $index_1 = index_2 = \dots = index_n = 1$ ;
4  while  $true$  do
5       $j = -1$ ;
6      for  $i = 1$  to  $n$  do
7          if  $index_i \leq length(l_i)$  then
8               $o = getElementAt(l_i, index_i)$ ;
9              if  $j = -1$  then
10                  $min = o$ ;
11                  $j = i$ ;
12             if  $o.t < min.t$  then
13                  $min = o$ ;
14                  $j = i$ ;
15         if  $j = -1$  then break;
16          $t = t \cup \{min\}$ ;
17          $index_j = index_j + 1$ ;
18 /* Synthesis of global log */
19  $log = []$ ;
20 for  $i = 1$  to  $length(t)$  do
21      $o_1 = getElementAt(t, i)$ ;
22      $o_2 = getElementAt(t, i + 1)$ ;
23     if  $o_1.act = SEND$  and  $o_2.act = RECEIVE$  and  $o_1.id = o_2.id$  then
24          $o = new Observation(null, o_1.id, null, o_1.s, o_1.r, o_1.m)$ ;
25          $log = log \cup \{o\}$ ;
26          $i = i + 1$ ;
27 return  $log$ ;

```

4.1.7 Tool & Experimental Evaluation

Test Case Experiment. We have experimented our approach and our tools on several medium-size case studies. For each of them, we have defined several correct implementations of its *Chor* choreography, and then we have performed mutations at the level of the implementation services representative of errors in the

Algorithm 7: Preorder Verification ($isPreorder(l, t)$)

Input: A trace $t = [\alpha_1, \alpha_2, \dots, \alpha_n]$
Input: A log $log = [ob_1, ob_2, \dots, ob_m]$
Output: **true** if $log \preceq t$ else **false**

- 1 **if** $m > n$ **then return false**;
- 2 **for** $i = 1$ **to** m **do**
- 3 **if** $ob_i.s \neq \alpha_i.s$ **or** $ob_i.r \neq \alpha_i.r$ **or** $ob_i.m \neq \alpha_i.m$ **then return false**;
- 4 **return true** ;

Algorithm 8: Global Conformance Verification

Input: A *Chor* specification C
Input: A set of n logs $\overline{log} = \{log_1, log_2, \dots, log_n\}$
Output: Verdict (**Pass** or **Fail**)

- 1 $T = \llbracket C \rrbracket$;
- 2 $l = \text{synthesisObservations}(\overline{log})$;
- 3 $l = l \downarrow_{act(C)}$; // get the global log
// hide additional interactions
- 4 **foreach** trace t of the T **do**
- 5 **if** $isPreorder(l, t)$ **then return Pass**;
- 6 **return Fail** ;

development/implementation process. These mutations can be categorized as follows: adding (a), removal (r), replacement (x), and reordering (o) of interactions, and change (c) of structuring operators.

Table 4.1: Online-Shopping Case Study

	Mutation	Chor specification			Observations		Verdict				Duration (seconds)
		# Roles	# Int.	# Traces	# 1	# 2	S_1	S_2	S_3	S	
1	–	3	8	5	5	4	✓	✓	✓	✓	0.013
2	–	3	8	5	16	14	✓	✓	✓	✓	0.017
3	–	3	8	5	16	14	✓	✓	✓	✓	0.018
4	(a) in S_3	3	8	5	17	14	✓	✓	✓	✓	0.016
5	(a) in S_2 & S_3	3	8	5	18	14	✓	✓	✓	✓	0.015
6	(r) in S_3	3	8	5	11	10	✓(✗)	✓(✗)	✓(✗)	✓(✗)	0.014
7	(r) in S_2 & S_3	3	8	5	14	12	✓	✓(✗)	×	×(✗)	0.014
8	(o) in S_2 & (c) in S_3	3	8	5	16	14	✓	×	×	×	0.018

We present in Table 4.1 results on one of our case studies, related to the online buying and delivery of goods (see below). The rows of the table correspond to different correct implementations (marked with –) and to different mutants. Its columns corresponds to the inputs which are a *Chor* specification defined by the number of: roles (# Roles), the number of interactions (# Int.) and the number of the traces (# Traces); and a set of logs represented by the number of observations before (#1) and after (# 2) the filtering on the set of interactions defined in the roles of the *Chor* specification (see Definition 9). The remaining columns are devoted to the kind of mutations being performed (Mutations), the testing verdicts (local, S_i , and global, S) and the duration of the testing process. As far as the verdicts are concerned,

an implementation or a service may be conform while not achieving completely the envisioned behavior. This may correspond to a potential deadlock situation. It is detected by our tool and denoted by (\boxtimes).

Our case study is described in *Chor* as follows:

$$C ::= \text{Order}^{[1,2]}; (\text{Reject}^{[2,1]} \sqcap \text{Confirm}^{[2,1]}; \text{Payment}^{[1,2]}; \\ (\text{Invoice}^{[2,1]} \mid \text{Shipment}^{[2,3]}; \text{Postage}^{[2,3]}; \text{Distribution}^{[3,1]}))$$

The implementation of this specification has been done with three Web services, Buyer (S_1), Vendor (S_2), and Shipper (S_3), running on three ODE engines.

The first three rows correspond to correct implementation logs. Row 1 corresponds to a case where the supplier rejects the order. It is hence much shorter, in observations, than rows 2 and 3. These correspond to the supplier accepting the order. The only difference between them is the order in which *Invoice* is done *wrt.* the rest of the choreography (*Shipment*, etc.), *i.e.*, the order in which parallel actions (\mid in the *Chor* specification) are done.

In row 4, a new message is sent in S_3 to inform S_2 about the goods being delivered to the client (*Inform*^{[3,2]!}). Since this is a new message *wrt.* the choreography, it is filtered out, and service S_3 is still conform to its role. The same yields for the whole implementation. In row 5, the corresponding message is also added in S_2 (*Inform*^{[3,2]?}). Again, it is filtered out and the implementation is correct. In row 6, S_2 is mutated in order not to send *Postage*^[2,3] to S_3 anymore. Since it is at the end of the S_2 role, this one is still conform to its role. Further, S_3 blocks waiting for it (hence it does not send *Distribution*^[3,1]), but also conforms to its role. Then, in row 7, both S_2 and S_3 have agreed not to use *Postage*^[2,3] (it is removed in both services). While S_2 stays conform, S_3 is not conform anymore, since it sends *Distribution*^[3,1] before having received *Postage*^[2,3], which is forbidden by its role.

Finally, we have a last mutation in row 8. We replace *Shipment*^{[2,3]!}; *Postage*^{[2,3]!} by *Postage*^{[2,3]!}; *Shipment*^{[2,3]!} in S_2 . We also replace *Shipment*^{[2,3]?}; *Postage*^{[2,3]?} by *Shipment*^{[2,3]?} *Postage*^{[2,3]?} in S_3 . Now S_2 and S_3 may interact doing *Shipment*^[2,3] before *Postage*^[2,3]. This is detected in the log of S_2 , the one of S_3 , and in the global log, hence all of these are not conform since this contradicts the specification.

We have experimented our approach on bigger case studies. The biggest one is a mutant with 7 services, 11 distinct interaction channels, and 116,640 traces in the choreography trace set. We have 52 observations (40 after filtering), and the testing time is 2.92 seconds. We have observed that the testing time is greater for fail

mutants since in the worst case we have to check all of the choreography traces to give a verdict (see Algorithm 8).

Generic Experiments. We have evaluated our testing approach by conducting practical experiments. Our aim was to assess the scalability (computation time) with reference to the number of messages, and this both for correct and incorrect logs. Our experiments can be explained as follows. We produce automatically a *Chor* specification by choosing the number of roles, the number of messages, and the operator types. Consequently, logs are produced automatically from this specification by applying the projection for each role. We then inject faults inside the logs in order to analyze the impact of the presence of faults on the time of the testing process. Different kinds of faulty logs (mutants) have been produced. In the experiments presented in Figure 4.7, we inject faults corresponding to the reordering of messages. The positions of faults are selected randomly. For sake of simplicity, we present here the experiments only for two services.

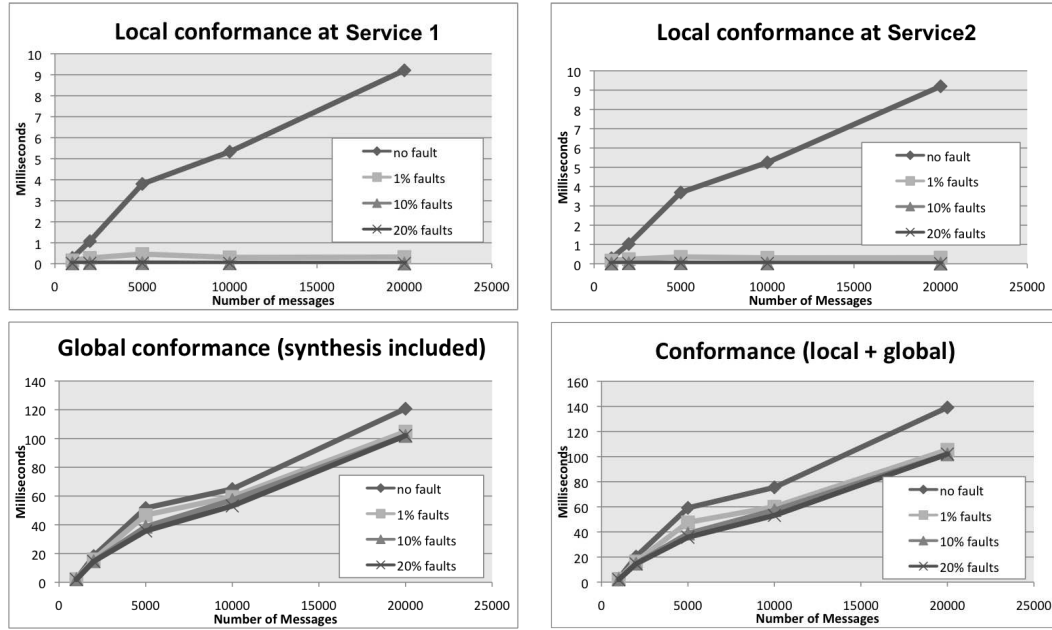


Figure 4.7: Tester Scalability

Figure 4.7 shows the time (in milliseconds) of verifications with 1,000, 2,000, 5,000, 10,000, and 20,000 messages, and for correct or incorrect logs. Each measure is computed from the average of 50 runs. Experiments are relative to the local verifications (for respectively service 1 and service 2), the global conformance verification (including global log synthesis), and the whole conformance testing process (sum of the 3). These experiments have been performed on a 2GHz Intel Core 2 Duo

MacBook laptop with 4GB of RAM. The times shown in Figure 4.7 do not take into account the time for the computation of the traces of the *Chor* specification since it depends only on this specification. For 1,000, 2,000, 5,000, 10,000, and 20,000 consecutive messages, this would be 43, 172, 1,099, 3,292, and 14,305 milliseconds respectively.

In Figure 4.7, we can observe that the local testing time is very small for any % of faults. In the worst case (upper bound), *i.e.*, for a correct log, we have to check the whole of it. The computation time for different % of faults in the global conformance testing is close to the case without faults. This is due to the important part of it used for the global log synthesis that grows with the number of messages. This directly impacts the overall testing time. Still, with a maximum time of 139 milliseconds for 20,000 messages, we believe that our approach is scalable.

4.2 Online Property-Oriented Testing

The conformance relation in the previous section is based on trace preorder relation, see Section 4.1.2. This relation allows to tests an implementation which realizes a prefix of choreography, *e.g.*, an implementation, which does nothing, always conform to its choreography. Moreover, this relation cannot test the fact that an event must happen after another. The work presented in this section will overcome these drawbacks. It focuses on testing some critical behavior, called property, rather than overall behaviors, of implementation. In this work, we assume that the properties are provided by the standards or by the choreography experts. They will be then checked on the execution traces, called *log*, of the IUT which are collected at running time. The formal model of the IUT is defined only to reason about the format of the logs and also about the one of properties. There is no need of the presence of model in testing process which is realized by our tool. However if the model is available, we have to check the correctness of properties *wrt.* the model to ensure no divergence between them.

This work intends to test the value-passing among implemented services. Since testing work on concrete data values presented in execution trace, in this approach, bounding variables by bound event, *e.g.*, $o^{[a,b]}? \langle x \rangle$, will be explicitly expressed by data assignments. We start by briefly introducing some notions about Symbolic Transition Graph with Assignments (STGA) which is used to specify distributed systems with a global perspective, *i.e.*, *choreographies*, and to describe the pieces of a distributed implementation, *i.e.*, *services*. A STGA is extended from STG to support assignments.

4.2.1 Symbolic Transition Graph with Assignments

Let \mathcal{D} be the, maybe infinite, set of data domains, ranged over by v , and \mathcal{V} be the finite set of variables, ranged over by x, y, z, x_1 , etc. We use $\text{dom}(x)$ to represent variable x domain, *i.e.*, $\text{dom}(x) \subseteq \mathcal{D}$. $DTerm$ is a set of data expressions, ranged over by t . $BTerm$ is a set of boolean expressions, ranged over by ϕ . We assume that $\mathcal{V} \cup \mathcal{D} \subseteq DTerm$, $t = t' \in BTerm$ for any $t, t' \in DTerm$. $BTerm$ is closed under the usual operators \wedge, \vee, \neg .

Definition 11 (Message & Event). *Given a finite set of operations \mathcal{O} , labels \mathcal{L} , and data domains \mathcal{D} , a message exchanging between two services of choreography takes the following form:*

$$o(l_1 = v_1, \dots, l_n = v_n)$$

where $o \in \mathcal{O}$ represents the name of the message and the composite data exchange is represented by a set $\{l_1 = v_1, \dots, l_n = v_n\}$, rewritten as $o(l_i = v_i)$ for short, in which each field of this data structure is pointed by a label $l_i \in \mathcal{L}$ and its value is $v_i \in \mathcal{D}$.

A basic event which represents the occurrence of a message $o(l_1 = v_1, \dots, l_n = v_n)$ takes the following forms, in which $v_i \in \text{dom}(x_i)$:

- Global level:
 - + interaction from role a to role b : $o^{[a,b]}(l_1 = x_1, \dots, l_n = x_n)$
- Local level:
 - + sending of interaction from role a to role b : $o^{[a,b]}!(l_1 = x_1, \dots, l_n = x_n)$
 - + reception of interaction from role a to role b : $o^{[a,b]}?(l_1 = x_1, \dots, l_n = x_n)$

A non-observable event, *e.g.*, internal computations, is noted by τ .

Listing 4.1: Example of Complex Message Exchange in XML

```

1 <Request>
2   <weight>3</weight>
3   <country>
4     <code>33</code>
5     <name>France</name>
6   </country>
7 </Request>
```

Example 10 (Message). We represent the Request in Listing 4.1 as following:

Request(weight = 3, country/code = 33, country/name = "France")

A STGA is a transition system where each state is associated with a set of free variables and each transition may be guarded by a boolean expression $\phi \in BTerm$ that determines if the transition can be fired or not. A *guarded transition* is labelled by a triple (ϕ, e, A) , e.g., $s \xrightarrow{[\phi] e/A} s'$ represents a guarded transition from state s to state s' with a guard ϕ , an event e , and an action A . An *action* A is a sequence of assignments. An *assignment* takes the form $x := t$. Thus, the action $A = (x_1 := t_1; x_2 := t_2; \dots; x_m := t_m)$, will be executed in a sequential manner. We denote A_x as $\{x_1, \dots, x_m\}$ and A as $\{t_1, \dots, t_m\}$. We use $fv(a)$ and $bv(a)$ to denote respectively the set of free and bound variables used in some expression a . These sets for an event will be detailed later.

Definition 12 (Symbolic Transition Graph with Assignments). *STGA is a tuple $\mathcal{M}(E) = (S, s_0, T)$ where, S is a non empty set of states, each state s having an associated set of free variables $fv(s)$, $s_0 \in S$ is the initial state, and T is a set of transitions. If $s \xrightarrow{[\phi] e/A} s'$, with $e \in E$, is a transition of T then $fv(\phi) \cup fv(e) \cup fv(A) \subseteq fv(s)$, $fv(s') \subseteq fv(s) \cup bv(e) \cup A_x$ and $bv(e) \cap (A_x \cup fv(A)) = \emptyset$*

In a transition $s \xrightarrow{[\phi] e/A} s'$, with $e \in E$, we can omit ϕ if it is true, and A if there is no assignment. In the above definition, neither $A_x \subseteq fv(s)$ nor $fv(e) \subseteq A_x$ is required.

Example 11. Let us present a running example with four services: c (client), s (shipping-quotation – Sq), a (accounting department – Ad), and b (bank center - Bc). Their STGAs are shown in Figure 4.8. The sets of free variables attached at initial states of these STGAs state that the client, the Sq, and the Ad services work with parameters $\{x_0, x_1\}$, $\{y_0\}$ and $\{z_0\}$ respectively, while the Bc service works without parameter. The client wants to ship some good, (s)he issues a request shipping to the Sq service by providing the weight of goods to be sent, then it receives a response indicating its price and a fee to pay. If the client agrees with this price, then the client will send its credit card number to the Sq service. In the Sq service side, after receiving the request, based on the received weight, and its price list (for sake of simplicity, we only consider two prices, 2 and 3), it will calculate a fee, then respond with the fee to the client. After that, it will wait to receive the client credit card number during one hour, then it will commit the client information to the Ad service. The Ad service will withdraw money from the Bc service based on the received information from the Sq service.

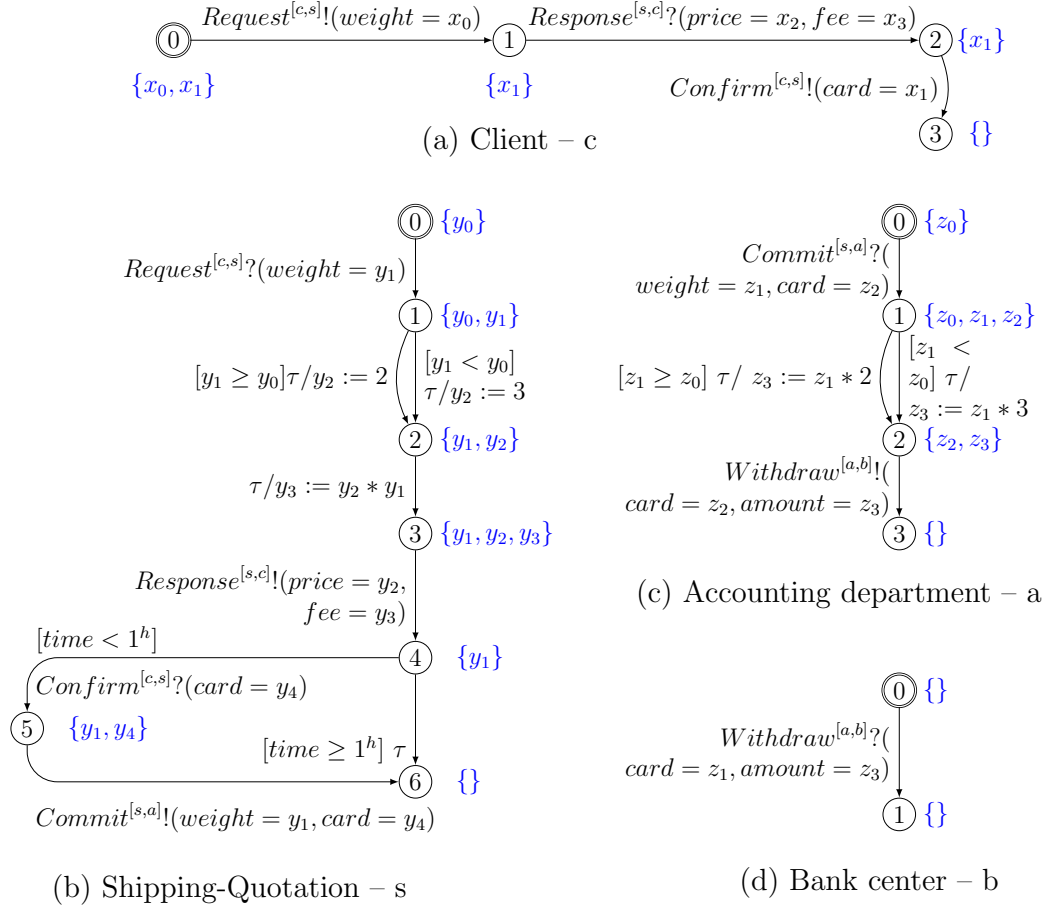


Figure 4.8: STGAs of four services

A shipping choreography, which presents the global behavior of the four services above, is as in Figure 4.9.

Let us note that the assignments presented in these STGs are used to represent the modifications of variables used by constraints of data carried by interactions. For example, in the global STG the weight in *Commit* interaction is the one in *Request* interaction while the fee responded by the *Sq* service to the client is either the double or the triple of the weight.

Semantics of STGA are introduced in [Li and Chen, 1999] by both symbolic and ground semantics. We choose late-ground semantics in our framework since we will work on implementation trace which is a sequence of messages. We concretize by using an evaluation function. An *evaluation* $\sigma \in Eval$ is a mapping from \mathcal{V} to \mathcal{D} . We denote $\sigma(t)$ as the evaluation of expression t by σ . Obviously, $\sigma(t) \subseteq \mathcal{D}$ and

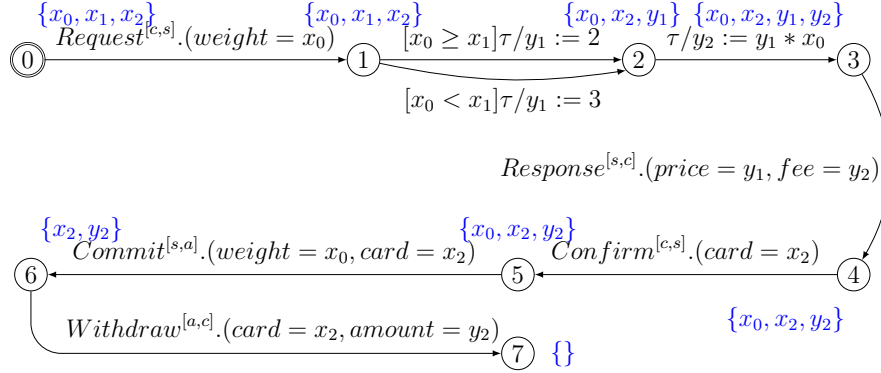


Figure 4.9: Shipping choreography

$\sigma(\phi) \in \{\mathbf{true}, \mathbf{false}\}$. We write $\sigma \models \phi$ to indicate $\sigma(\phi) = \mathbf{true}$. Composition of evaluations σ and σ' is denoted by $\sigma.\sigma'$ such that $\sigma.\sigma'(t) = \sigma(\sigma'(t))$.

A (finite) *path*, π , of an STGA $\mathcal{M}(E)$ is a sequence of consecutive transitions in $\mathcal{M}(E)$. *Runs of a path*, $\sigma(\pi)$, is created by applying σ on π as depicted by rules in Figures 4.10. Its result is a sequence of ground transitions where there is no guard, *i.e.*, it is true, each state consists of a state of STGA, an evaluation, and concrete event, called message in which each of its parameter is a constant v . A message m is a structure $o(l_1 = v_1, \dots, l_n = v_n)$ where o is the control part and $\bigcup \{l_i = v_i\}$ represents the data part. A *trace* is a sequence of messages created from a $\sigma(\pi)$ by removing all the states without changing the messages order.

Definition 13 (Trace Semantics of STGA). *The semantic of a STGA $\mathcal{M}(E)$ is the set of all its traces $\llbracket \mathcal{M}(E) \rrbracket$.*

$$\begin{array}{cc}
\begin{array}{c}
(\text{SEND}) \\
\frac{s \xrightarrow{[\phi] \ o!(l_i=x_i)/A} s'}{s_\sigma \vdash \xrightarrow{o!(l_i=\sigma(x_i))} s'_{A.\sigma}} \sigma \models \phi
\end{array}
&
\begin{array}{c}
(\text{INTERACT}) \\
\frac{s \xrightarrow{[\phi] \ o.(l_i=x_i)/A} s'}{s_\sigma \vdash \xrightarrow{o.(l_i=\sigma(x_i))} s'_{A.\sigma}} \sigma \models \phi
\end{array} \\
\\
\begin{array}{c}
(\text{RECEIVE}) \\
\frac{s \xrightarrow{[\phi] \ o?(l_i=x_i)/A} s'}{s_\sigma \vdash \xrightarrow{o?(l_i=v_i)} s'_{A.\sigma \cup \{x_i \mapsto v_i\}}} \sigma \models \phi
\end{array}
&
\begin{array}{c}
(\text{TAU}) \\
\frac{s \xrightarrow{[\phi] \ \tau/A} s'}{s_\sigma \vdash \xrightarrow{\tau} s'_{A.\sigma}} \sigma \models \phi
\end{array}
\end{array}$$

$\forall v_i \in \text{dom}(x_i)$

Figure 4.10: Trace Semantics of STGA

Example 12 (Trace). $m = \text{Response}^{[s,c]}!(\text{price} = 2, \text{fee} = 6)$ is a message of the *Sq* service model in Figure 4.8(b). A trace of this model is

$$\langle Request^{[c,s]}?(weight = 3), \tau, \tau, Response^{[s,c]}!(price = 2, fee = 6) \rangle,$$

while the trace below is not

$$\langle Request^{[c,s]}?(weight = 3), \tau, \tau, Response^{[s,c]}!(price = 2, fee = 10) \rangle$$

In the sequel of the Section, we consider only observable messages, *i.e.*, they are not τ messages, and observable traces, *i.e.*, they contain only observable messages, since those are what we observe from the execution of an IUT. Indeed, from the running IUT, we cannot observe internal activity representing by τ in the model. An *observable trace* is obtained from $\llbracket \mathcal{M}(E) \rrbracket$ by removing τ events while preserving the order of other messages. $\llbracket \mathcal{M}(E) \rrbracket$ is overridden as the set of all (observable) traces of model $\mathcal{M}(E)$. *Log*, also called *execution trace*, recorded from an IUT will have the format of an observable trace.

4.2.2 Local Properties

In this section we formally define properties. A property represents behaviors the IUT is expected to satisfy. Since these behaviors to be tested are usually far fewer than the behaviors of IUT, this approach reduces not only a lot of processing, but also allows the tester to focus on critical behaviors of the IUT [Li et al., 2004, Ladani et al., 2005]. We define *local property* \mathcal{P} to express a behavior to be tested at the level of one service, while *global property* $\bar{\mathcal{P}}$ is used to express collaborations and/or relation between data among services to be tested. The negative version of properties is also presented to guarantee that the IUT *does not* perform some special behaviors which can lead to erroneous behaviors. These properties are then checked against the execution logs of the IUT to emit a testing verdict. Local property checking requires only local log of the corresponding service, while a set of local logs are required for global one.

Our property is expressed as a IF-THEN clause, *e.g.*, *IF context THEN consequence*, *i.e.*, each time a *context* is satisfied then the *consequence* must appear. For example, each time the Sq service in Figure 4.8 receives a *Request* from the client, then it must respond. Furthermore, message exchanges carry information which can be validated under some condition described by a boolean expression, *e.g.*, the *fee* response must be equal to the multiplication of the *price* response and the requested *weight*.

A message is an instance of an event (under an evaluation), *i.e.*, an event expresses a set of messages which have the same operation name and set of labels, *i.e.*, which differ at the values. We use candidate event, which is a pair event/predicate, e/ϕ , to represent a sub-class containing messages which are instances of the event e that are satisfied by ϕ .

Definition 14 (Candidate Event). A *Candidate Event (CE)* is a pair $o(l_1 = x_1, \dots, l_n = x_n)/\phi(x_1, \dots, x_n)$, denoted by $o(\bar{x})/\phi(\bar{x})$, where $\phi(x_1, \dots, x_n)$ is a boolean expression on $\{x_1, \dots, x_n\}$. A CE $o(\bar{x})/\phi(\bar{x})$ is called to be validated by message $o'(l'_i = v_i)$ iff $o' = o \wedge \bigwedge l_i = l'_i \wedge \rho \models \phi$, with $\rho = \bigcup \{x_i \mapsto v_i\}$.

By extension, we write $o(\bar{x})/\phi(\bar{x}_1, \bar{x})$ to present that this CE may depend on another CE $o_1(\bar{x}_1)/\phi(\bar{x}_1)$. The predicate can be omitted if it is true.

Example 13. $CE_1 = \text{Request}^{[c,s]}?(weight = x)/(x > 0)$ expresses a class of received Request message of the Sq service from the client such that the value of the requested weight parameter is positive.

Definition 15 (Local Property). Local property P is described by the form:

$$\begin{aligned} \mathcal{P} &::= \text{Context} \xrightarrow{(d)} \text{Consequence} && (\text{positive}) \\ \neg \mathcal{P} &::= \text{Context} \xrightarrow{(d)} \neg \text{Consequence} && (\text{negative}) \end{aligned}$$

where

- d is a positive integer,
- *Context* is a sequence of CEs, e.g., $\langle e_1(\bar{x}_1)/\phi_1(\bar{x}_1), \dots, e_n(\bar{x}_n)/\phi_n(\bar{x}_1, \dots, \bar{x}_n) \rangle$, and
- *Consequence* is a set of CEs, e.g., $\{e'_1(\bar{y}_1)/\phi'_1(\bar{x}_1, \dots, \bar{x}_n, \bar{y}_1), \dots, e'_m(\bar{y}_m)/\phi'_d(\bar{x}_1, \dots, \bar{x}_n, \bar{y}_m)\}$.

This definition allows to express that each time when the *Context* is satisfied then the *Consequence* must or must not (depending on the formula type, i.e., \mathcal{P} or $\neg \mathcal{P}$) be validated after at most d messages. Since it is impossible to verify online the fact that will occur in the future. The distance d can become also from requirements, e.g., an account should be blocked if user enter 5 consecutively wrong passwords. The *Context* is satisfied when all of its CEs are satisfied while the *Consequence* is satisfied when there exists at least one CE which is satisfied. The *Consequence* is not satisfied when all of its CEs are not satisfied.

Example 14. Let us take some examples of properties:

$$\begin{aligned}
P_1^s &::= \langle Request^{[c,s]}?(_) \rangle \xrightarrow{(1)} \{Response^{[s,c]}!(_) \} \\
P_2^s &::= \langle Request^{[c,s]}?(weight = y_1) \rangle \xrightarrow{(1)} \{Response^{[s,c]}!(price = y_2, fee = y_3) / (y_3 == y_1 * y_2) \} \\
P_1^a &::= \langle Commit^{[s,a]}?(weight = x_1, card = x_2) \rangle \xrightarrow{(1)} \{Withdraw^{[a,b]}!(card = y_1, amount = y_2) / \\
&\quad (y_1 == x_2 \wedge (y_2 == x_1 * 2 \vee y_2 == x_1 * 3)) \}
\end{aligned}$$

P_1^s guarantees that the Sq service always responds to its received request. P_2^s details the property P_1^s by adding a predicate which guarantees that the computation of fee in the Sq service is correct. P_1^a guarantees that the Ad service transfers exactly the credit card number from the Sq service to the Bc service and there are two prices 2 and 3 which can be applied.

Negative property is introduced as the reverse of positive one. If a model of the IUT is available, we can easily obtain positive properties which corresponds to a negative one and vice versa. A negative property corresponding to P_1^s is:

$$\neg P_{11}^s ::= \langle Request^{[c,s]}?(_) \rangle \xrightarrow{(1)} \neg \{Confirm^{[c,s]}?(_), Commit^{[s,a]}!(_) \}$$

This property states that the Sq service has not to receive a *Confirm* or send a *Commit* immediately after receiving a *Request* from the client.

Verification of Local Property on Log. Given a (potentially infinite) log $log = \langle m_1, \dots, m_i, \dots \rangle$, and a property $P = \langle e_1/\phi_1, \dots, e_n/\phi_n \rangle \xrightarrow{(d)} \{e'_1/\phi'_1, \dots, e'_m/\phi'_m\}$, we define the semantics, *i.e.*, the returned verdict, of the property P on the log log by means of an algorithm. The algorithm works as follows. It is based on a *Check* function which takes as inputs the execution log and a property to be checked. In a property, a later CE may depend on a former one, consequently, verification of a message may require the presence of its precedence. Since we can forward-only read data in a continuous stream mode, we need to create buffer which contains some fragment of messages stream, what we call a *window*. The created windows contain the first message validating the first CE of the context property and the following messages the $n + d$ next messages. Once a window is created, the verification process on the window can start in parallel with the other created windows. In case of a positive property, the verdict **Fail** is emitted only when no message in the d -next messages of the log satisfies any CEs of *Consequence*. The expected verdicts (**Pass** or **Fail**) can be given only when the *Context* is validated. An **Inconclusive** verdict is emitted only if the *Context* cannot be matched, *i.e.*, there is no **Pass** or **Fail** verdicts are emitted. The algorithm needs to collect minimum $n + d$ messages for validating the property (n messages for *Context* and then d messages for *Consequence*). There are maximally $(n + d) + (n + d - 1)$ messages registered in memory.

4.2.3 Global Property

The local property is used to express some behavior of a service and is tested by using only log of the service itself. However, *some kinds of fault cannot be detected by local property*. Let us take an example by considering the number precision problem in the Ad service of our running example as following. When the Ad service received $Commit^{[s,a]}?(card = x, weight = y)$ from the Sq service, value of y will be rounded to one digit, *e.g.*, the Sq service sends $weight = 4.96$ but the Ad service will consider the received weight as 5. In consequence, if the Ad service is configured with $z_0 = 5$, the price 2 will be applied instead of the price 3. This kind of fault in the Sq service can be detected since its response contains *price*, *e.g.*, by property P_2^s . But, the *Withdraw* event sent by the Ad service does not contain the *price*, this cannot be detected by verifying the relation between *amount* and *weight*. However as required by the choreography model in Figure 4.9, the price applied by the Ad service has to be the one applied by the Sq service, *i.e.*, if price 3 is applied by the Sq service then it must be also applied by the Ad service and the same holds with price 2. In such a case, local log of only the Ad service is not sufficient. We need to analyze several local properties on a global log to detect such a fault.

The global log of the choreography IUT was constructed by the synthesis of services local logs of a choreography by assuming the existence of a global clock [Zaidi et al., 2009], *e.g.*, the IUT is running in a cloud. The global clock allows to know the total order of messages from the set of local logs. In this work, our *global log* is just constructed by grouping local logs in a set. Since no synthesis is required, we do not need a global clock.

Definition 16 (Global Property). *The global property is described wrt. the following grammar:*

$$\overline{\mathcal{P}} ::= SET \Longrightarrow SET' \mid \neg \overline{\mathcal{P}} ::= SET \Longrightarrow \neg SET'$$

where SET and SET' are two set of local properties.

Verification of Global Property on Global Log. We firstly formalize global log as follows:

Definition 17 (Global Log). *Let log_1, \dots, log_n be n local logs recorded from n different services. We define the global log, \overline{log} , of these n services as the set of their logs. Local log of service i in global log is given by $\overline{log}|_i$.*

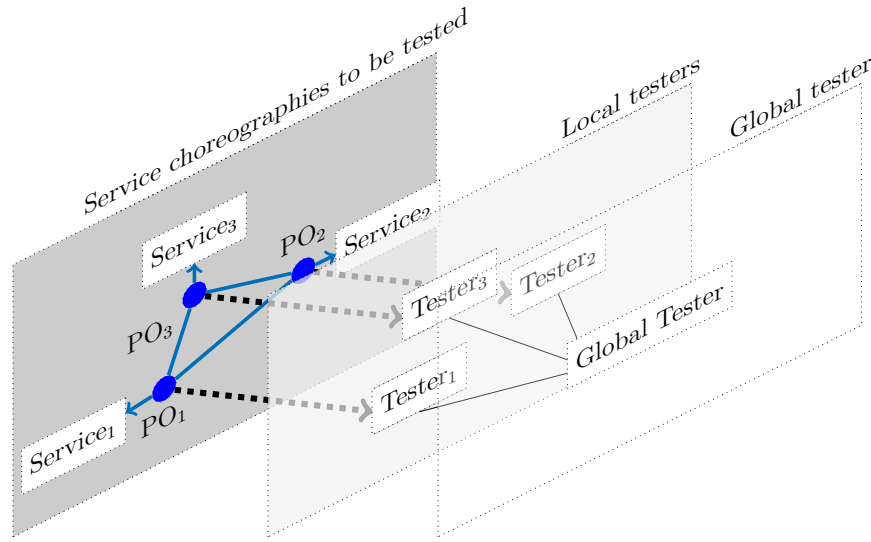


Figure 4.11: Architecture of the Verification System

the property is validated. The **Inconclusive** verdict of the property will be emitted by the tester when the end of stream of the log is reached without any delivered verdict.

- Step 4: **Extraction of execution traces.** An observer is put at each service level to sniff all of its (input and output) messages exchanged with its partners. Each time the observer captures a message, if the message is related to the properties to be tested, then it is sent through an opened pipeline between the tester and the PO to the tester, where it will be verified by an XQuery processor.
- Step 5: **Properties tested on the execution traces.** The properties tested in XQuery form will be executed by MXQuery⁶ processor on the XML stream supplying by the observer. Based on result of the query, the verdict (**Pass**, **Fail**, or **Inconclusive**) will be emitted.

In this verification, the steps 1, 2 and 3 are done once, while the step 4 and 5 are done in a continuous way online. The Figure 4.12 exhibits the steps 3, 4, and 5. In the sequel, we will present all the steps of the framework in the order, except that step 4 will be explained before step 3 since the translation into XQuery depends on the format of the log file.

⁶<http://mxquery.org>

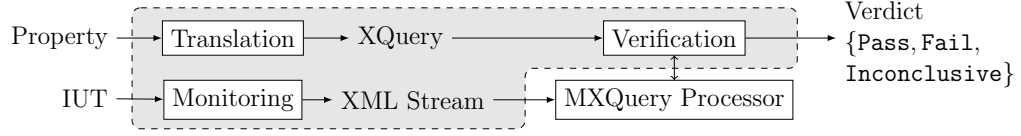


Figure 4.12: Online Verification of Local Property

Correctness of a Local Property. When a model of a service is available, we suppose that the service (the implementation) must conform to its model, *i.e.*, the logs of the service must conform to its traces. The correctness of local property *wrt.* a service model is guaranteed by the following definition.

Definition 18 (Correctness of Local Property). *Let \mathcal{M}^S be a service model and P be a property, we say P is correct (return true) *wrt.* \mathcal{M}^S if:*

- $\forall tr \in \llbracket \mathcal{M}^S \rrbracket, Check(tr, P) \in \{Pass, Inconclusive\},$ and
- $\exists tr \in \llbracket \mathcal{M}^S \rrbracket, Check(tr, P) = Pass$

Since we cannot always compute all traces of \mathcal{M}^S , *i.e.*, it may be infinite due to the unboundedness of the model equipped with loops. To overcome this issue, we put limit k which will cut the length of paths. For the data types, we avoid the state space explosion by avoiding the unfolding of the model, the obtained paths are symbolic paths, *i.e.*, no concrete values are given to variables. The verification of the correctness is performed by the Algorithm 9. We firstly find all paths of \mathcal{M}^S being compatible with the events of the *context* of P . Two events are compatible iff they have the same control part and list of parameter labels. For each found path, we add the predicates of each CE of P to the guard of the transition of the compatible event with the one of the CE if this is a sending, while for the reception the predicate is added in the next transition. One of the two special transitions are added to each path, lines 21 and 34, to distinguish if the path is or not compatible also with the *consequence* of P . Z3 SMT solver is used, line 37–38, to determine whether the cumulated predicates are satisfiable, *i.e.*, it exists a set of instances (values) for the variables of the path.

Correctness of Global Property. Since the global property consists of local properties whose correctness are verified on local traces, we override the projection function “ \downarrow ” applied for projecting global trace to local traces.

Definition 19 (Trace Projection). *Let \mathcal{M}^C be a choreography model, and $tr \in \llbracket \mathcal{M}^C \rrbracket$. We override the projection of tr on service a as:*

Algorithm 9: Checking Correctness of Local Property

Data: an STGA $\mathcal{M} = (S, s_0, T)$, a number $k > 0$, and a Property

$$P = \langle e_1/\phi_1, \dots, e_n/\phi_n \rangle \xrightarrow{(d)} \{e'_1/\phi'_1, \dots, e'_m/\phi'_m\}$$

Result: true/false (property P correct/incorrect)

```

1  Get all paths  $\Pi_1$  of  $\mathcal{M}$ : each starts from  $s_0$ , and its  $n$  last observable events are compatible with
    $\langle e_1, \dots, e_n \rangle$ , and its length isn't greater than  $k$ ;
2  if ( $\Pi_1 == \emptyset$ ) then return false;                                     // not found any path
3   $\Pi_3 := \emptyset$ ;
4  foreach  $\pi_1 \in \Pi_1$  do
5       $\pi'_1$  is minimum path at the end of  $\pi_1$  compatible with  $\langle e_1, \dots, e_n \rangle$ ;
6       $\varphi := \text{true}$ ;     $k := 1$ ;
7      // for each CE in context
8      foreach transition  $tr_j = s \xrightarrow{[\phi] e/A} s' s \xrightarrow{[\phi] e/A} s' \in \pi'_1, j = 1$  to  $\text{length}(\pi'_1)$  do
9          if  $e$  is compatible with  $e_k$  then
10              $\varphi := \varphi \wedge \bigwedge (x_i == y_i)$ , with  $e$  is  $o(l_i = x_i)$  and  $e'_k$  is  $o(l_i = y_i)$ ;
11             if  $e$  is reception then
12                 update  $tr_j$  with  $\phi := \phi \wedge \varphi$ ;
13                  $\varphi := \phi_k$ ;
14             else
15                 update  $tr_j$  with  $\phi := \phi \wedge \phi_k \wedge \varphi$ ;
16                  $\varphi := \text{true}$ ;
17              $k := k + 1$ ;
18   $s \xrightarrow{[\phi] e/A} s'$  is the last transition of  $\pi$ ;                                     //  $\Rightarrow e \equiv e_n$ 
19  Get all paths  $\Pi_2$  of  $\mathcal{M}$ : each starts from  $s'$ , its length is not greater than  $d$ , and the its last
   event is in  $\{e'_1, \dots, e'_m\}$ ;
20  if ( $\Pi_2 == \emptyset$ ) then
21       $\Pi_3 := \Pi_3 \cup \{\pi_1 \wedge (s' \xrightarrow{[\varphi] \times} s_\times)\}$ ;
22  else foreach  $\pi_2 \in \Pi_2$  do                                     // for each CE in consequence
23       $k := 1$ ;
24      foreach transition  $tr_j = s \xrightarrow{[\phi] e/A} s' \in \pi_2, j = 1$  to  $\text{length}(\pi_2)$  do
25          if  $e$  is compatible with  $e'_k$  then
26              $\varphi := \varphi \wedge \bigwedge (x_i == y_i)$ , with  $e$  is  $o(l_i = x_i)$  and  $e'_k$  is  $o(l_i = y_i)$ ;
27             if  $e$  is reception then
28                 update  $tr_j$  with  $\phi := \phi \wedge \varphi$ ;
29                  $\varphi := \phi_k$ ;
30             else
31                 update  $tr_j$  with  $\phi := \phi \wedge \phi'_k \wedge \varphi$ ;
32                  $\varphi := \text{true}$ ;
33              $k := k + 1$ ;
34       $\Pi_3 := \Pi_3 \cup \{\pi_1 \wedge \pi_2 \wedge (s'' \xrightarrow{[\varphi] \checkmark} s_\checkmark)\}$ ;
35   $b := \text{false}$ ;
36  foreach  $\pi_3 \in \Pi_3$  do
37      if exist an evaluation  $\rho$  s.t. state  $s_\times$  exists in  $\rho(\pi_3)$  then return false;
38      else if exist an evaluation  $\rho$  s.t. state  $s_\checkmark$  exists in  $\rho(\pi_3)$  then  $b := \text{true}$ ;
39  return  $b$ ;

```

$$\begin{aligned}
- \quad \text{proj}(o^{[a,b]}.(l_i = x_i), d) &= \begin{cases} \langle o^{[a,b]}!(l_i = x_i) \rangle & \text{if } d = a \\ \langle o^{[a,b]}?(l_i = x_i) \rangle & \text{if } d = b \\ \langle \rangle & \text{otherwise} \end{cases} \\
- \quad tr|_a &= \begin{cases} \langle \rangle & \text{if } \text{length}(tr) = 0 \\ \text{proj}(\text{head}(tr), a) \frown \text{tail}(tr)|_a & \text{otherwise} \end{cases}
\end{aligned}$$

Definition 20 (Correctness of Global Property). *Given a choreography model \mathcal{M}^C , a set of n service models $\{\mathcal{M}_1^S, \dots, \mathcal{M}_n^S\}$, and a global property $\bar{P} = SET \mapsto SET'$. \bar{P} is correct wrt. \mathcal{M}^C iff:*

- $\forall P_a \in SET \cup SET'$ of service a having model \mathcal{M}_a^S , $\exists tr \in \llbracket \mathcal{M}_a^S \rrbracket$, $\text{Check}(tr, P_a) = \text{Pass}$
- $\forall tr \in \llbracket \mathcal{M}^C \rrbracket$, $\text{Check}(tr, \bar{P}) \in \{\text{Pass}, \text{Inconclusive}\}$
- $\exists tr \in \llbracket \mathcal{M}^C \rrbracket$, $\text{Check}(tr, \bar{P}) \in \{\text{Pass}\}$

In the definition above, the two last conditions can be also verified based on Algorithm 9 where the notion of a compatible event is modified as follows. A sending or reception event is compatible with an interaction if it is compatible with the projection of the interaction.

Extraction of Execution Traces. We extend our monitor presented in Section 4.1.4 to collect SOAP messages exchanged among Web services of a choreography. Each service is attached by one observer at a point of observation such that it can capture all SOAP messages from and to its service. We then put body parts of captured message side by side as shown in Example 16 in which, we add also *tstamp* attribute representing time stamp of the capturing moment. This *tstamp* allows us to verify time condition, *e.g.* timeout condition in our example where after the *Response* event the *Confirm* event can only happen before one hour.

Example 16. Listing 4.2 represents the captured log of the *Sq* service.

Listing 4.2: Example of Captured Message

```

1 <message source="c" destination="s" direction="reception" name="Request"
  tstamp="1">
2   <weight>3</weight>
3 </message>
4 <message source="s" destination="c" direction="sending" name="Response"
  tstamp="3">
5   <price>2</price>    <fee>6</fee>
6 </message>
7 ...

```

Translating Property into XQuery. The SOAP messages exchanged between Web services are in XML format. One can refer to record execution traces (*log*) as an XML document to take advantage of standardized XML tools, *e.g.*, XML Query Language (XQuery), to analyze it. XQuery is a language for finding and extracting elements and attributes from XML data. The log of the IUT can be considered as a data stream consisting of continuous messages with time-varying arriving and unpredictable rates. Hence, the log processing requires real-time treatment, fast mean response time, and low memory consumption. We use window clauses in XQuery to slice the log into segments called window. The translation is done automatically, by Algorithm 10, in which two functions $ctrl(e)$ and $cond(\phi)$ are used to translate the event e into XQuery and the assertion ϕ into XQuery respectively.

Algorithm 10: Translation of Local Property to XQuery

Data: Local property $P = \langle e_1/\phi_1, \dots, e_n/\phi_n \rangle \xrightarrow{(d)} \{e'_1/\phi'_1, \dots, e'_m/\phi'_m\}$
Result: an XQuery

```

1  return XQuery as the following:
2  for $w in (
3    for sliding window $win in $stream//message
4    start $s at $spos when  $ctrl(e_1)$  and  $cond(\phi_1)$ 
5    end $e at $epos when  $\$pos - \$spos \text{ eq } (n + d - 1)$ 
6    return <window> { $win } </window> ) return
7  for $e1 in $w/message[1] return
8  for $e2 in $w/message[2] where  $ctrl(e_2)$  and  $cond(\phi_2)$  return
9  ...
10 for $en in $w/message[n] where  $ctrl(e_n)$  and  $cond(\phi_n)$  return (
11   (some $e11 in $w/message[position() > n] satisfies  $ctrl(e'_1)$  and  $cond(\phi'_1)$ ) or
12   ...
13   (some $elm in $w/message[position() > n] satisfies  $ctrl(e'_m)$  and  $cond(\phi'_m)$ ))

```

Example 17. The query in Listing 4.3 represents a local property of Example 14. Line 2–5 creates a sequence of windows, each window is represented by a variable \$win. The variable \$stream points to a log in stream mode and \$stream//message is used to denote all message elements in the log. A window \$win is a sub sequence of \$stream for which the start and end conditions are applied. XQuery uses XPath syntax to express specific parts of an XML element. Our window starts at a message such that its destination is the Sq service, *i.e.*, “s”, its name is “Request”. The window size is 2. The size can be less than 2 if we reach the end of \$stream. Each created window realized by an XQuery from line 2–5 is encapsulated by another XQuery and will be referenced by \$w, which has in charge to perform the checking of the property of the created window (line 7–8). For the example, at line 7 it verifies whether a message in window \$w has its position which is strictly greater than 1 and that the conditions in line 8 are satisfied. If both holds a *Pass* verdict is emitted.

Listing 4.3: Example of Transformation of Local Property into XQuery

```

1 for $w in (
2   for sliding window $win in $stream//message
3   start $s at $spos when $s/@name eq "Request" and $s/@direction eq "
      reception" and $s/@receiver eq "s"
4   end $e at $epos when $epos - $spos eq 1
5   return <window>{$win}</window> ) return
6 for $e1 in $w/message[1] return
7   (some $e11 in $w//message[position() > 1] satisfies
8     $e11/@name eq "Response" and $e11/@source eq "s" and $e11/@direction
      eq "sending" and number($e11/fee) eq number($e11/price) * number($e1
      /weight) )

```

4.2.5 Experimental Evaluation

To evaluate the performance of our tools, we realized a series of experiments. For the verification of each property in Example 14, we generate 50 logs files for each length 1,000, 2,000, 5,000, and 20,000 messages. Each of these logs contain a randomly created sequence of messages corresponding to the shipping-quotation service and the Ad service for the test of properties P_1^s , P_2^s , and P_1^a respectively. Generated logs are then sent as an XML stream to our tool, as described in Figure 4.12.

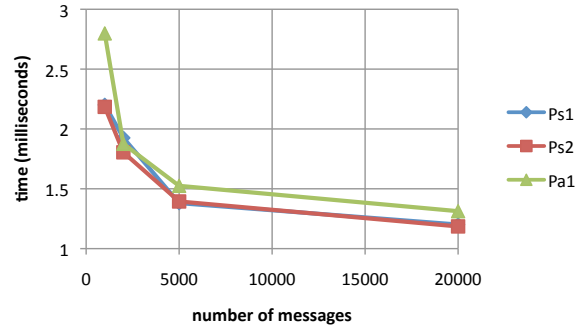


Figure 4.13: Tester Scalability

Since the performance of our tool depends on the one of the MXQuery processor, we do not intend to benchmark MXQuery processors, but rather to get early experiences of online verification of our approach. Generally, processing time of a property depends on the complexity of its boolean expressions. Since the processing time consists of time used to start up MXQuery and useful time used to validate the logs, the average processing time will converge on the useful time. Figure 4.13 shows the average of processing time (milliseconds). These experiments have been realized on a Macbook Air laptop with a CPU 1.7GHz Core i5 and 4GB of RAM. These results show that the maximum processing time per message are 2.8 milliseconds. This time will converge on the useful time, that is less than 1.4 milliseconds, when length of

logs increases. This tends to show that our framework can actually be done in real time.

4.3 Discussion

Passive testing is suitable for testing distributed system. It does not require to control IUT as active testing. It performs the testing by passively observing the inputs and outputs of IUT. Consequently, it does not disturb the functioning of the IUT. There exist two passive testing approaches: naïve and property-oriented. The naïve approach examines any observations while the property-oriented approach focuses only on observations which relate to some critical behavior of IUT. This chapter presented our works based on these approaches. We firstly started to test offline service choreographies. This work, based on the naïve approach, does not take into account value-passing in the choreography. This was the first step to establish the elements for passive testing of distributed system, *e.g.*, interpretation of asynchronous communication by five cases of observation correlations, monitoring of SOAP messages. The limitations of the first work was overcome by the second one. The second work focuses on online-testing, *i.e.*, it detects the faults as soon as possible, and it does not require a global clock. However the second one may miss some faults due to the lack of property expressiveness, *e.g.*, local property cannot verify backward and global property cannot verify correlations of messages which are from different local logs. These lacks can be fulfilled by the first approach.

Our works presented in this chapter is based on the verification of execution traces of IUT, *i.e.*, they are done after the IUT was deployed. They detect faults which already occurred. It may be preferred to test the IUT before it is deployed to be able to repair the IUT. In such a case, the IUT can be tested separately, *i.e.*, each service. Consequently active testing has more advantages than passive testing to test one single service. Test cases to test actively each service can be generated from local requirements which are projected from the choreography by our projection defined in Section 3.4.

Case Study

Contents

5.1	Case Study Description	100
5.2	Verification	100
5.3	Testing	105

This thesis has proposed a symbolic framework, which is fully tool supported, for modeling, verification and testing service choreographies. For each step, the implementation and evaluation of corresponding tool was presented in the previous chapters. We demonstrate in this chapter, through a simple case study, an application of our toolchain to support choreography development process: modeling, verification and testing. The toolchain was written in around 34.000 lines of Java. It is available for downloading or online use, *e.g.*, Symbolic Choreography Analysis tool (SChorA).

5.1 Case Study Description

The case study in this chapter is an extended version of the Online Shopping Process (OSP) presented Example 1 of Chapter 3. The case study can be modified gradually to point out attached problems. For instance, it represents in the next section a collaboration between three roles: *buyer*, *vendor* and *warehouse*. First the *buyer* sends to *vendor* a request by indicating an amount to be bought. If this amount is greater than 0 then the *vendor* forwards the request to the *warehouse*, otherwise the *vendor* raises an error to the *buyer*. After receiving request from *vendor*, the *warehouse* checks also the amount. It responds if the requested amount is greater than 0, otherwise it raises also an error.

5.2 Verification

In this section, we verify the case study by using our SChorA tool. The verification consists of reachability checking, realizability checking, choreography projection, and conformance checking.

5.2.1 Reachability Checking

Figure 5.1 represents specification of the case study and reachability checking by using the SChorA tool. The case study specification is written in editor part of SChorA. For sake of simplicity, we denote *b* as the *buyer*, *v* as the *vendor*, and *w* as the *warehouse*. It respects our language presented in Section 3.1. A specification is encapsulated into a component by **component** and **end component** keywords. A component can be also given by an STG. There may be many components in the editor part. They are put after **DECLARATIONS** keyword. The declared components are verified by using corresponding commands that are put after **COMMANDS** keyword, *e.g.*, **showSTG spec** displays graphically STG of the **spec** component. The reader is invited to refer to Appendix A for details of syntax of SChorA.

The reachability checking of the **spec** component is done thanks to **showReachableSTG spec** command. The reachable STG is shown at the bottom of Figure 5.1.

In this STG, the $(5) \xrightarrow{[x \leq 0] \text{ error}^{[w,v]}} (3)$ transition was removed. Indeed, input verification of the *warehouse* is dispensable in the case study choreography since the input was already verified by the *vendor* before transferring it to the *warehouse*. In other words, this transition is never fired, hence unreachable, since the disjunction of its guard, $x \leq 0$, and the guard of its precedent transition, $x > 0$, is always **false**.

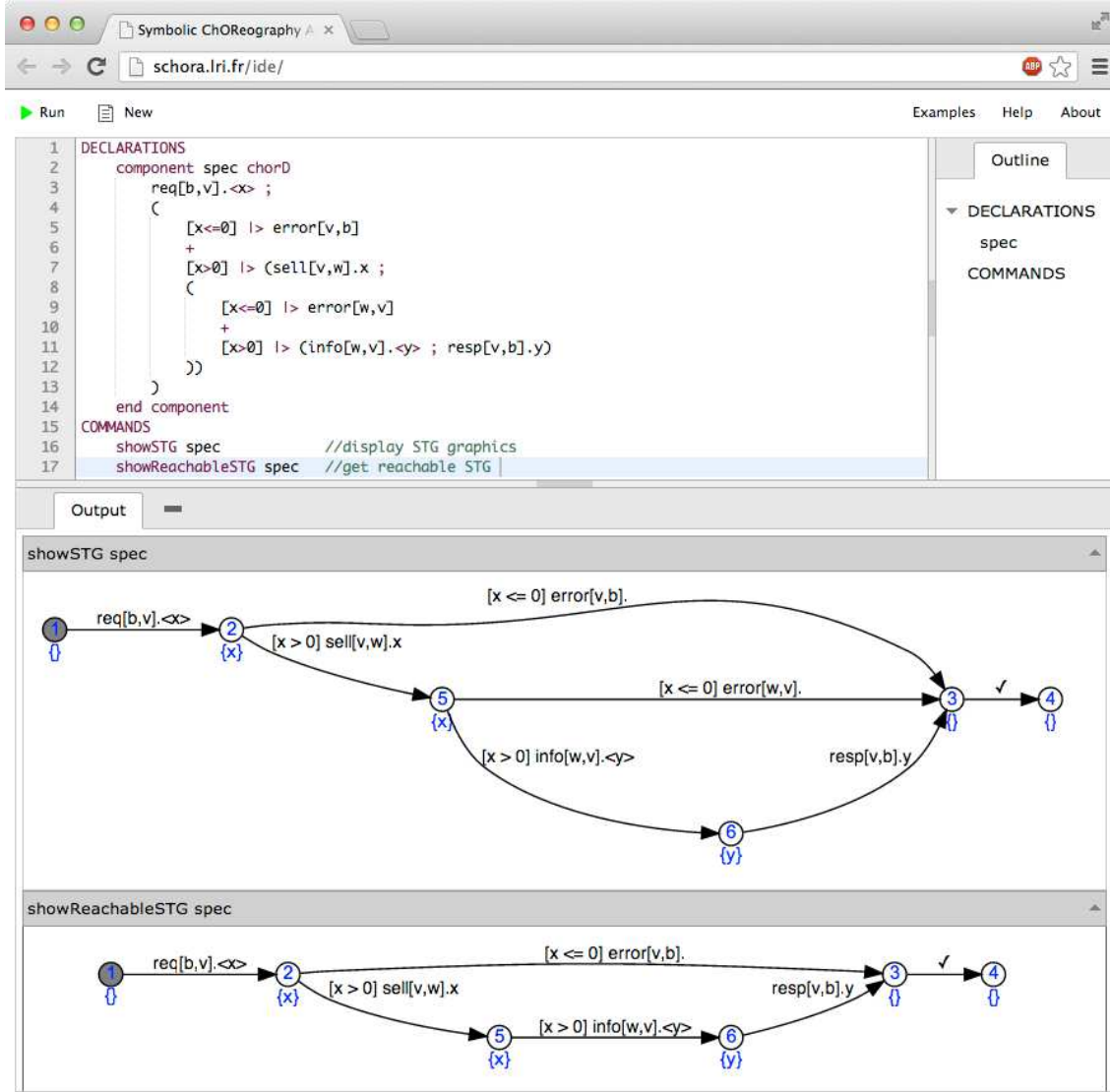


Figure 5.1: Reachability Checking by SChorA tool

5.2.2 Realizability Checking

We check the realizability, under synchronous communication mode, of the case study choreography by using `showRealizableSTG spec SYNC` command. The result is presented in Figure 5.2. The choreography is not realizable since it needs an additional interaction $+cb^{[v,w]}.x$. The interaction is introduced to transfer value of x variable from the *vendor* to the *warehouse*. Indeed, if the *buyer* sends a request with a negative number to the *vendor* then the choreography will be terminated after the interaction $error^{[w,v]}$. In such a case, the *warehouse* does not know that the choreography was terminated. It may be blocked if it chooses another branching of the choreography, *e.g.*, it waits the $sell^{[v,w]}.x$. To render the choreography realizable, the *warehouse* should know which branching of the choreography is selected. This can be done by an additional interaction, *e.g.*, $+cbr^{[v,w]}.x$ which carries value of x , hence the *warehouse* selects branch to follow based on value of x .

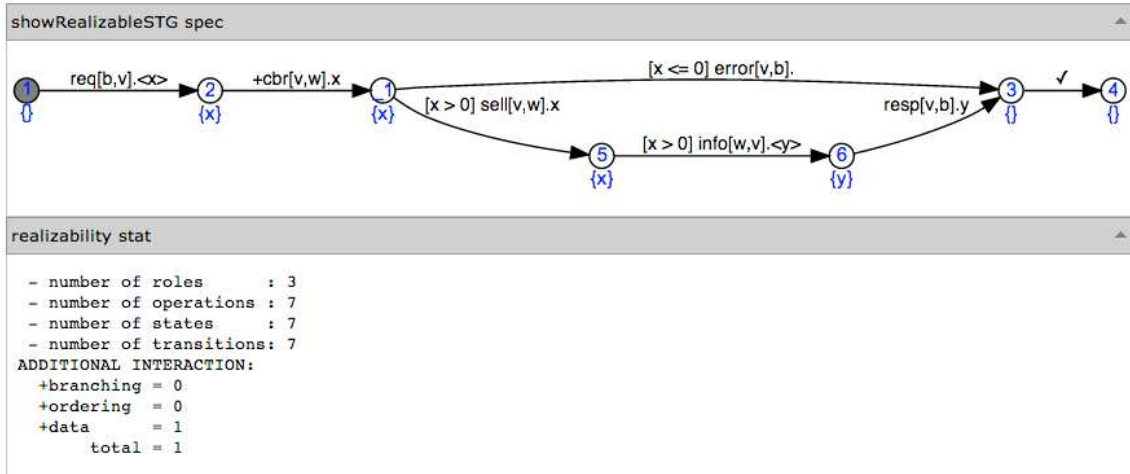


Figure 5.2: Realizability Checking by SChorA tool

5.2.3 Choreography Projection

Projection under synchronous communication mode of the case study choreography is done by `projection spec SYNC` command. Its result is three local STGs of three roles as presented in Figure 5.3.

We would like to point out the interesting branching decision of state (6) of obtained local STG of the *buyer* (role b). Indeed, the guards of the outgoing transitions of this state was removed since the branching can be decided by event, *e.g.*, the *buyer* chooses the route when it receives $error^{[v,b]}?$ or $resp^{[v,b]}?$.

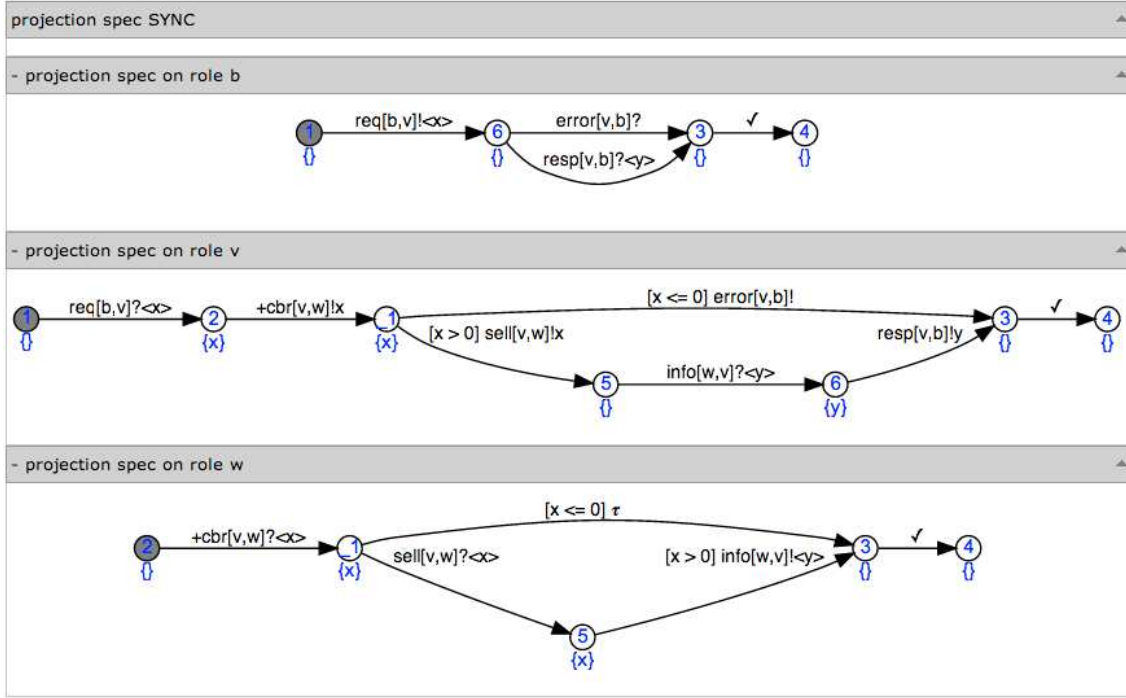


Figure 5.3: Choreography Projection by SChorA tool

5.2.4 Conformance Checking

The conformance checking verifies the Symbolic Branching Bisimulation for Conformance (SBBC) relation between two global STGs, one for implementation and one for specification. Figure 5.4 presents an example of conformance checking of an implementation of the case study. Let us suppose that one implements the case study based on the local models obtained by the projection in Figure 5.3. These local models of three services *buyer*, *vendor* and *warehouse* are described as three components, from line 6 to 15, in the editor part of SChorA in Figure 5.4. The implementation of the case study choreography is the composition of the three services. It is represented by the `impl` component. The verdict **Pass** of the result in Figure 5.4 means that the implementation conforms to the choreography. The result also gives the Predicate Equation System of this relation. This proves once again that the projection is correct, *i.e.*, each service can be separately implemented from its local model.

Let us suppose that the *warehouse* service is partially implemented, *e.g.*, it always waits the interaction $cbr^{[v,w]}$ from the *vendor*. The STG composition of local models of this *warehouse* service with the two above services, the *buyer* and the *vendor* is presented in Figure 5.5. The conformance checking by SChorA shows that this

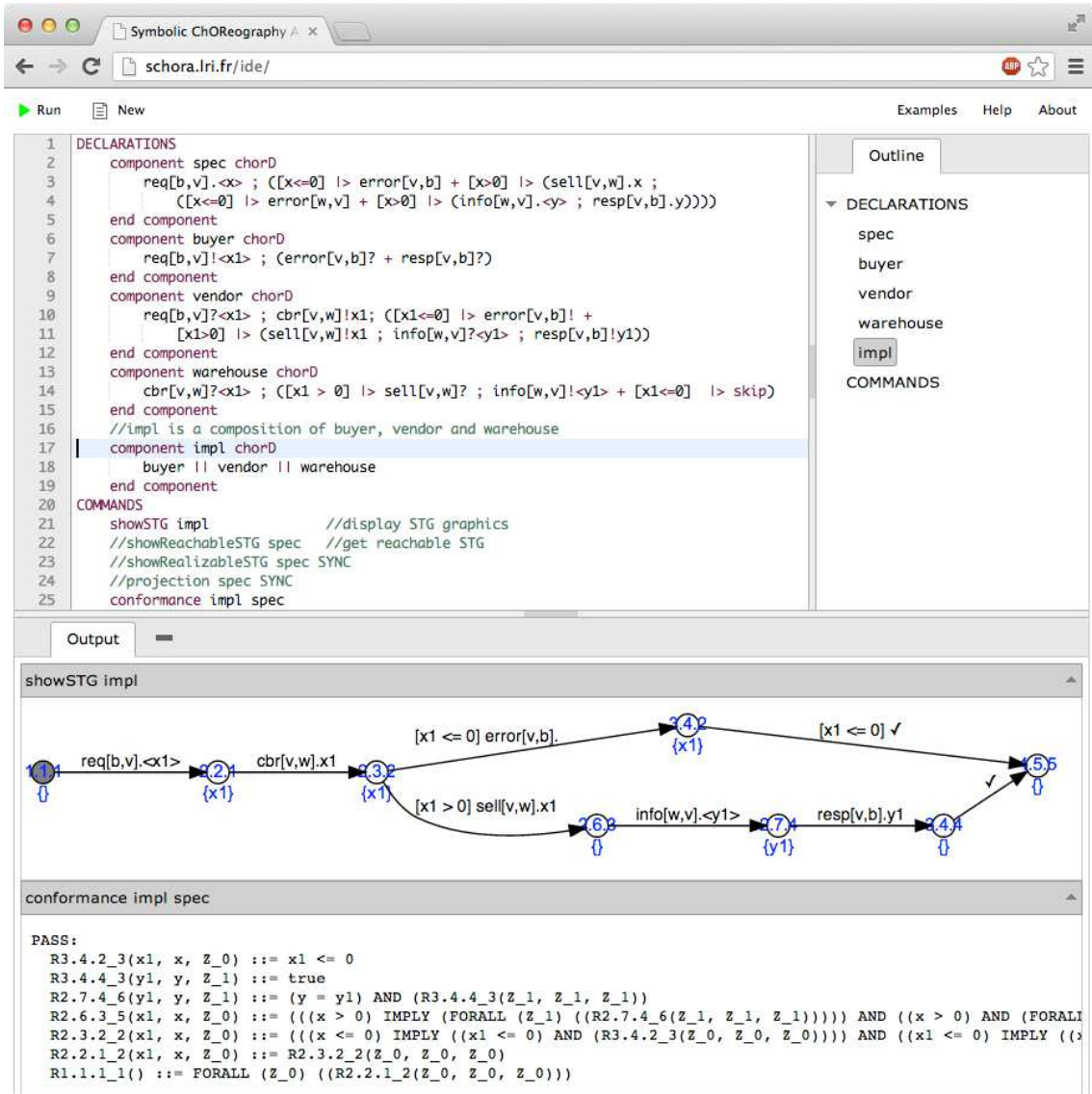


Figure 5.4: Conformance Checking by SChorA tool

implementation does not conform to the choreography, *i.e.*, **Fail** verdict. Indeed, thanks to the Predicate Equation System (PES) of this conformance relation, we can see that the non-conformance is due to state (3.4.2) of the implementation composition and state (3) of the specification (see Figure 5.2), *e.g.*, $R3.4.2.3 ::= \text{false}$. From the state (3) the **spec** terminates properly by doing \checkmark event however the **impl** cannot evolve anymore from the state (3.4.2).

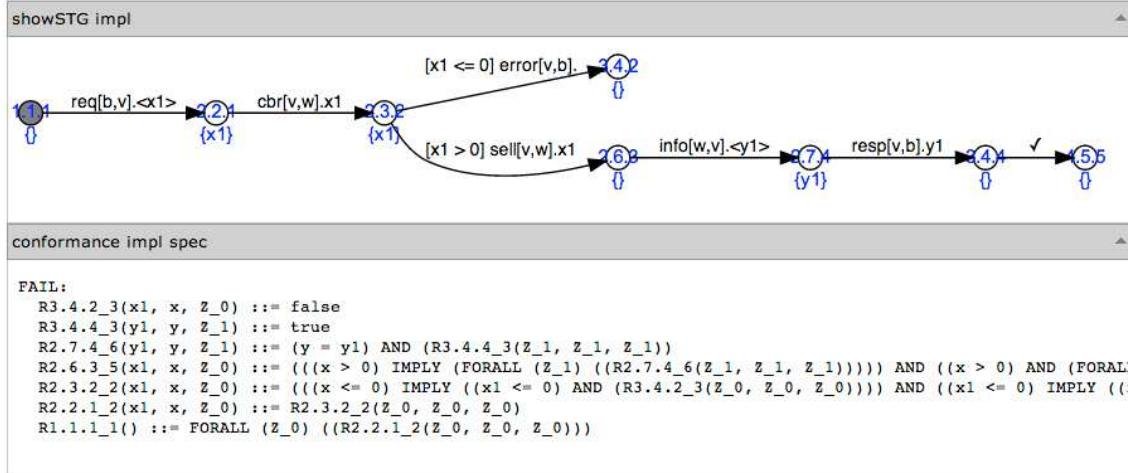


Figure 5.5: Conformance Checking of a Mutation of the Case study

5.3 Testing

From the case study choreography **spec**, the projection is done to obtain local requirements of each role, see Figure 5.3. Developers need to implement services which fulfill these requirements. This section presents two tools to test a real implementation, that is a set of Web services, with respect to the choreography **spec**. The first tool tests conformance of the implementation, while the second one focuses on some critical behavior of the implementation.

We constructed two tools, SOAP-Capturer and SOAP-Forwarder that are used to capture SOAP messages transferred among Web services. The captured messages can be saved in a file to be used later, or be broadcasted in order to be analyzed in real time by our tester tools, *e.g.*, Property-Oriented Testing tool (Prop-tester), or by any standardized XML tools, *e.g.*, XQuery.

SOAP-Capturer is a module functioning inside Apache ODE¹ which is a WS-BPEL compliant web service orchestration engine. It captures any SOAP message received

¹<http://ode.apache.org/>

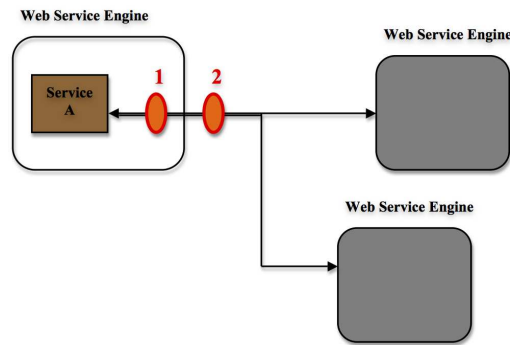


Figure 5.6: Point of Observation of SOAP-Capturer (1) and SOAP-Forwarder (2)

or sent by the Apache ODE. SOAP-Forwarder is a standalone application which is put among the sender and receiver services to capture SOAP messages transferred between them. It has a graphic user interface as shown in Figure 5.7.

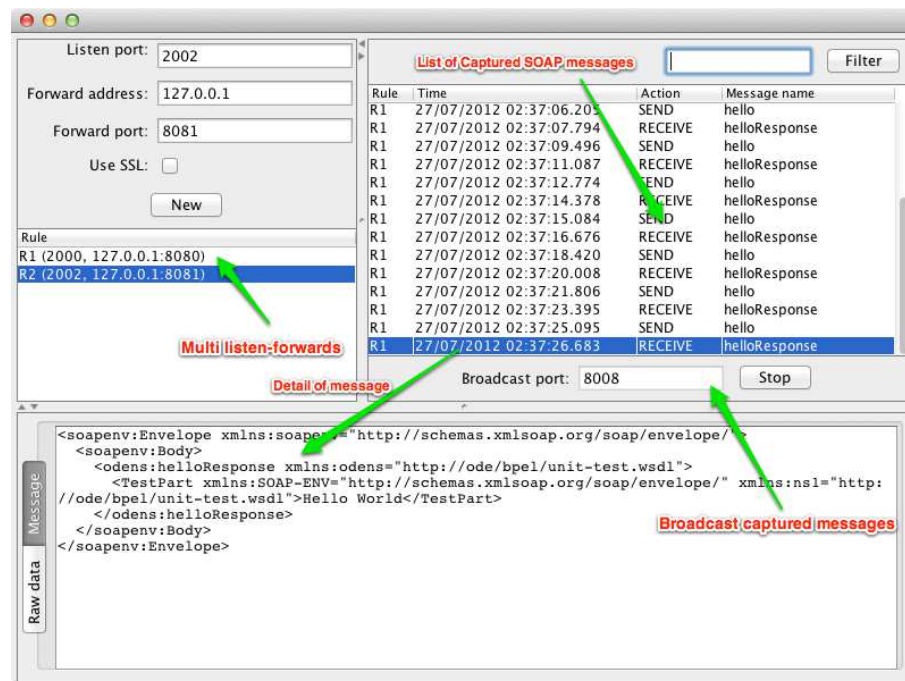


Figure 5.7: Graphic User Interface of SOAP-Forwarder

5.3.1 Conformance Testing

The conformance testing is done by Passive Conformance Testing tool (PACT) tool. The tool takes as input two parameters, a choreography description and a list of

logs (execution traces) of services which implement roles of the choreography. The choreography is described in *Chor* [Qiu et al., 2007] language, see Section 4.1.1. *Chor* does not support value-passing. The case study, after removing unreachable events, is described by *Chor* language as the following, in which role 1 is the *buyer*, role 2 is the *vendor* and role 3 is the *warehouse*: `req[1,2] ; (error[2,1] [] (sell[2,3] ; info[3,2] ; resp[2,1]))`

The logs of services are recorded by our monitoring tool, see Section 4.1.4. Listing 5.1 represents a log of vendor service which implements the *vendor*. Vendor service is a Web service which is written in WS-BPEL language. It is deployed into Apache ODE engine at address `http://localhost:8081/ode/processes/vendor`. We use address of service to identify the service.

Listing 5.1: Example of Captured log of Vendor Service

```

1 http://localhost:8081/ode/processes/vendor
2
3 29/08/2013 15:17:30.128
4 RECEIVE|http://localhost:8080/ode/processes/buyer|1377782250120|req
5
6 29/08/2013 15:17:30.320 (1377782250320)
7 SEND|http://localhost:8082/ode/processes/warehouse| |sell
8
9 29/08/2013 15:17:30.678
10 RECEIVE|http://localhost:8082/ode/processes/warehouse|1377782250634|info
11
12 29/08/2013 15:17:30.721 (1377782250721)
13 SEND|http://localhost:8080/ode/processes/buyer| |resp

```

A captured event is represented by two lines in log file, *e.g.*, line 3-4 in Listing 5.1. The first one indicates the moment the event is captured. The second one contains: event type (sending or reception), partner address, correlation identification, and event operation name. The correlation identification is used to synthesize global log from local logs of services, *e.g.*, a sending and its reception is synthesized as an interaction. Since we suppose that there exists a global (logic) clock among the implemented services, the correlation identification is based on sending moment of event. For instance, the identification 1377782250120 of the reception `req` at line 4 is the sending time of the sending `req` that can be found in the log of buyer service.

The conformance testing of an execution of implementation which consists of three Web services: buyer service, vendor service, and warehouse service which play respectively role 1, 2 and 3 of the choreography is done by PACT tool as presented in Figure 5.8.

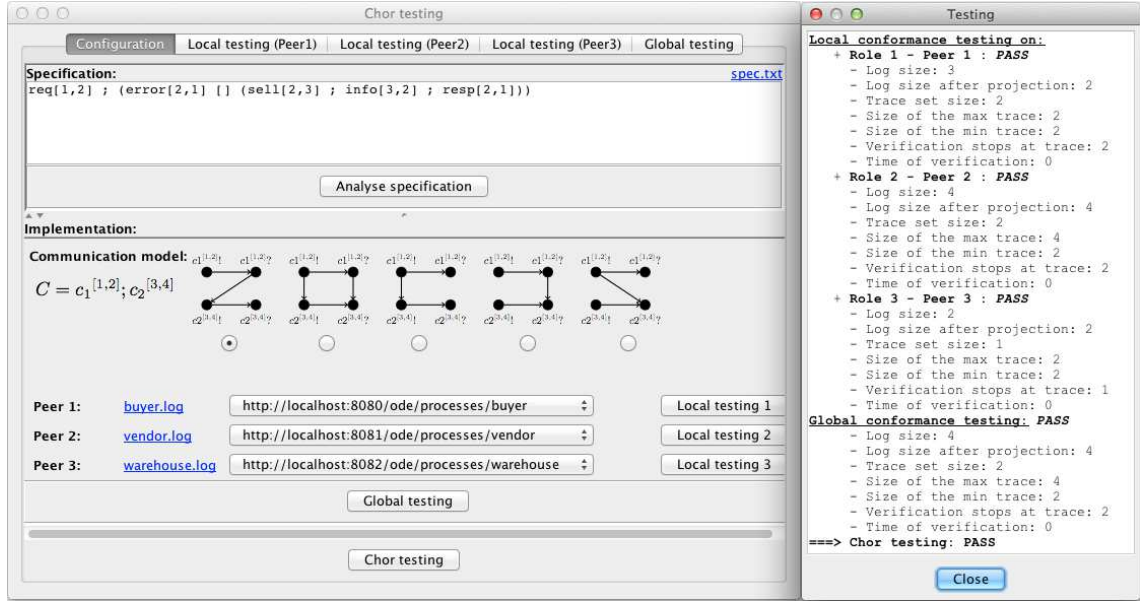


Figure 5.8: Conformance Testing of an Implementation

5.3.2 Property-Oriented Testing

Choreography implementation can also be tested by Prop-tester. It focuses on testing some critical behavior of the implementation rather than overall one as by PACT. The critical behavior to be tested is described by a property. If model of implementation or of specification are available, the properties must conform to the model. This can be done by a module of Prop-tester in which the model is given by a STGA.

Property Description. Property definitions are described in XML files. Each property is described in `<property></property>` tag. Property type can be **positive** or **negative**. Namespaces used in properties are declared in `<namespace></namespace>` tag. The boolean expression of each candidateEvent is a string *wrt*. XPath syntax, described in `<predicate></predicate>` tag.

Local property described in Listing 5.2 verifies that when the vendor service receives a request `req` with a positive `amount` from the buyer service, then this amount must be equal to the `amount` of interaction `sell` to the warehouse service. Furthermore, the vendor service must perform `sell` interaction after receiving `req` and before any other interactions.

Listing 5.2: A Local Property to Verify Vendor Service

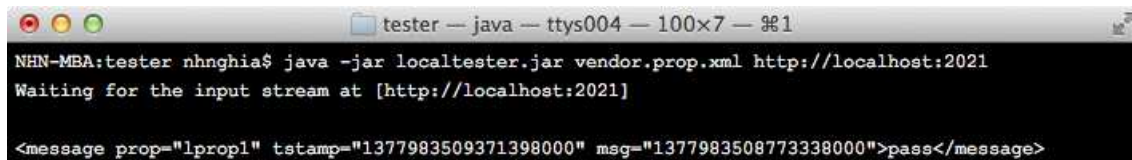
```
1 <properties>
```

```

2      <!-- Vendor Service transfers exactly what it received from the
        buyer to the warehouse-->
3      <namespaces>
4          <namespace prefix="tns" uri="http://www.lri.fr/" /> </namespaces>
5      <property name="lprop1" type="positive" distance="1">
6          <context>
7              <candidateEvent>
8                  <event>
9                      <data>
10                         <x1>from</x1>
11                         <x2>to</x2>
12                         <x3>tns:req/tns:amount</x3> </data> </event>
13                     <predicate><![CDATA[
14                         x1 == "http://localhost:8080/ode/processes/buyer"
15                         and x2 == "http://localhost:8081/ode/processes/vendor"
16                         and x3 > 0]]></predicate> </candidateEvent> </context>
17                 <consequence>
18                     <candidateEvent>
19                         <event>
20                             <data>
21                                 <y1>from</y1>
22                                 <y2>to</y2>
23                                 <y3>tns:sell/tns:amount</y3> </data> </event>
24                             <predicate><![CDATA[x3 == y3
25                                 and y1 == "http://localhost:8081/ode/processes/vendor"
26                                 and y2 == "http://localhost:8082/ode/processes/warehouse"]]>
27                             </predicate> </candidateEvent> </consequence>
28                 </property>
29 </properties>

```

Verdict Output. Prop-tester output its verdicts in continuous stream mode, *i.e.*, a verdict is emitted each time a property is tested on the log. A verdict is given in an XML element `<vdict/>` which has 3 attributes: `prop` the property name to be tested, `tstamp` the testing time, and `msg` the time stamp of tested message which validated the first event candidate of the property, *i.e.*, if verdict is `Fail` then it represents the moment of fault occurring. Figure 5.9 represents output verdicts of testing local property `lprop1` on the vendor service.



```

tester — java — ttys004 — 100x7 — %1
NHN-MBA:tester nhnghia$ java -jar localtester.jar vendor.prop.xml http://localhost:2021
Waiting for the input stream at [http://localhost:2021]
<message prop="lprop1" tstamp="1377983509371398000" msg="1377983508773338000">pass</message>

```

Figure 5.9: Verdict Output of Vendor Service Testing

Conclusions and Perspectives

Contents

6.1	Synthesis of Results	112
6.2	Perspectives	114

6.1 Synthesis of Results

In this thesis, we proposed a theoretical and practical framework which supports choreography development process. Particularly, we obtained the following results:

Identification of Lack of Value-Passing Support. The first contribution of the thesis relies on the observation of lack of value-passing support in modeling and analyzing service choreographies. Most existing approaches do not adequately support the value-passing, *i.e.*, without explicitly considering the data exchanged through interactions and using it for branching decisions. They just *abstract away from data*. This may yield *over-approximation issues*, *e.g.*, false negatives in verification process. Hence the presence of value-passing in choreography model may change choreography properties, *e.g.*, realizability, conformance. The lack of value-passing support becomes the principal motivation of the thesis.

A Symbolic Model & Framework for Choreographies. Based on the identification of lack of value-passing, we enriched the interaction-based model by introducing free and bound variables at global and local models. Based on calculus languages presented in [Bravetti and Zavattaro, 2007, Busi et al., 2006, Qiu et al., 2007] we formalized a process algebra language using for both choreography and implementation (services) specification. This language describes not only interactions but also data flow. We proposed a fully symbolic semantics, using STG, of the model in which, the data variables are manipulated by using symbols rather than their concrete values. This avoids data abstraction and over-approximation problems. Furthermore, it enables one to model and analyze service choreography in presence of data without suffering from state space explosion and without bounding data domains.

Based on the formal model, we re-examine several fundamental issues of services choreography with the presence of value-passing such that the conformance, the realizability and the projection. A new issue examined is *reachability* since with the presence of data in the formal model, the guard operations based on data, *e.g.*, \triangleright (if) and $*$ (loop), may disable some interactions in the choreography, see Example in Figure 2.9(a).

Accordingly, we build on branching bisimulation [Van Glabbeek and Weijland, 1996] and on a symbolic extension of weak bisimulation [Li and Chen, 1999] to develop a specific *symbolic version of branching bisimulation* dedicated at *checking the conformance* of a set of local entities *wrt.* a choreography specification. Our equivalence enables one to check conformance in presence of choreography *refinement*, *i.e.*, where new services and/or interactions may be added *wrt.* the specification. Going further than a *true* vs. *false* result for conformance, our approach supports the generation of the *most general constraint over exchanged information* in order to

have conformance.

We did not only check whether the choreography is realizable, but also we proposed solutions to enable its realizability. For that purpose, we built a *projection function*, dealing with data, which allows to retrieve local models from a global one. The projection is dedicated to enable the realizability of choreography, *i.e.*, when the choreography is unrealizable, some *extra interactions* can be automatically introduced. Furthermore, the minimum extra interactions are added in order to obtain minimal implementation and traffic. The projection considers both *synchronous and asynchronous communication modes*.

Passive Online Testing Approach. In distributed context, the control overall IUT is difficult to achieve, it is even impossible in some situation, *e.g.*, system working 24/7 where the IUT can be running in their real environment. Therefore, the testing *should not disturb* its natural operation, since it might produce a wrong behavior of the overall system.

We defined a formal model based on STGA [Li and Chen, 1999] for both local (service, role) and global (choreography, composition of local models) with supporting complex data types. We formalized our local and global properties, inspired from [Andrés et al., 2010] by taking data into account. The local properties are used to test behaviors of one isolated service *wrt.* its specification model, while the global properties test the collaboration of a set of services *wrt.* its choreography model. A negative version of property is also introduced. A *positive property* is used to describe expected behavior, while a *negative property* describes a forbidden behavior. If a model is available, we should verify the correctness of these properties *wrt.* its model. Once the correctness of the properties is ensured, both local and global properties are verified against the local execution traces of the running IUT collected at each service. Our verification process does *not require a global clock* since we assume that global properties express relation between several local properties and in particular by specifying relation between data among the choreography. We validated the approach by a case study in which services are realized by Web services. The SOAP messages exchanged of each service are progressively collected as a XML stream. We translated our properties into XQuery to perform an *online verification* of the properties on the IUT XML stream.

Implementation of the Proposed Approaches. The last contribution of this thesis is the availability of our tools. They demonstrate the proposed methods. The first tool, namely SChorA, allows to analyze some choreography attributes such that reachability, realizability, to check conformance of a set of local models against a choreography model, and to project a choreography model to local models with enabling the realizability attribute (in the case the choreography is unrealizable).

An interested feature of this tool is that one can use it directly from web browser without any installation and configuration.

Two tools, PACT for conformance testing and Prop-Tester for property-oriented passive testing, performs passive testing of service choreography. The Prop-Tester is an improvement on the PACT. Particularly, it first takes into account value-passing in testing process. Secondly, it focuses on some critical behavior of IUT rather than overall one. Thirdly, it performs online test, *i.e.*, at running time. Finally, it does not require the presence of a global clock on IUT.

Our case studies was implemented by Web services. We hence also developed two tools to collect SOAP messages transferring among Web services. One tool, namely SOAP-Capturer, is executed inside Apache ODE environment. Since the tool is only suitable for WS-BPEL Web services, we then developed another tool, SOAP-Forwarder, which functions independently from Web service engines. All the tools are freely available to download, use, and develop.

6.2 Perspectives

In this section, we present some existing issues and discuss some open directions.

True-Concurrency Supported. A lack of the formal framework proposed in this thesis is true-concurrency. Since choreography intends to specify collaboration of a set of services which are usually implemented in a distributed environment, *e.g.*, Web services, hence the true-concurrency is an inevitable attribute. This attribute is not appropriately supported by the current approaches. It may be also need to re-examine the fundamental issues of choreography when this attribute is present in the model.

Application of the Proposed Framework to Diagnose Service Choreographies. Conformance checking or conformance testing intends to determine whether a fault occurs rather than locates where the fault occurs. In environment of service collaboration, by locating services which is cause of faults, one can isolate the rest of the system from the faults. The fault services can be repaired or replaced basing on their local models which are generated from the choreography by using our projection.

Resolving Mismatch for Service Choreography. Using a bottom-up development process, each service of a distributed collaborative application may be independently developed and deployed across different organizations. It is therefore unavoidable that mismatch may arise at both signature and protocol levels, and need to be identified and solved. When two or more services are incompatible, an adaptor

may be introduced to solve mismatch [Mateescu et al., 2012]. The adaptor runs parallel with the services and guides their execution, *e.g.*, avoiding deadlock of their composition. In combination with adaptation, our approach can be used to resolve mismatch between reused services and between these services' composition and the requirements expressed by the choreography. Skeletons constitute local contracts that services have to fulfill to yield an overall composition being correct *wrt.* the choreography. Given each of these skeletons, and the corresponding service one wants to reuse for it, we can generate an adaptor between the skeleton and the service. This can be performed in a fully automatic way for some mismatch. For the more powerful adaptation approaches, *i.e.*, for more complex mismatch, the process can be aided [Cámara et al., 2012].

Generation of Source Code for Service Implementations. In a top-down development process, once the behavioral skeletons have been generated from the choreography with our approach, one has to implement the business logic behind the protocols expressed in the skeletons. In order to get full compatibility between the business logic and the behavioral protocols, one may use results presented in [Pavel et al., 2005, Fernandes and Royer, 2007, Fernandes et al., 2007] to retrieve Java code from skeleton models. This is made possible since the formal models we use for skeletons, STG, are very close to the Symbolic Transition System (STS) used in the above-mentioned approaches

Generation of Distributed Test Cases. Furthermore, we will study the distributed testing of choreography implementations based on the generation of test cases or properties from the local model obtained by projecting choreography. Role models can be used to generate test cases. Many approaches support the production of tests from symbolic transition models. In [Bentakouk et al., 2009], the authors generate symbolic test cases according to different coverage criteria or test purposes. A product is performed between the test purpose and the symbolic model. A symbolic execution tree for this product is generated and used to obtain symbolic test cases, *i.e.*, test cases with free variables and constraints on their values. These test cases are then realized and executed against the implementation with the use of a constraint solving tool. Role models generated with the approach we propose yield global conformance with the choreography. In such a case, the interaction part is guaranteed, in other words testing in isolation each service implementing a role can be sufficient to warranty the correction of the choreography implementation.

Transformation Standard Choreography Languages to STG. This concerns the definition of model transformations from standard choreography languages, *e.g.*, WS-CDL and BPMN 2.0, to our choreography model. For instance, since there exists a transformation [Bentakouk et al., 2009] from WS-BPEL, an execution language for

Web service orchestration, to STS. Hence if this transformation is done, then the tool chain of conformance verification of choreography and its Web service implementation by WS-BPEL will be done completely automatically.

Integration of Tools in Eclipse. Our tools are freely available. They can be download and used at local computer or directly from a web browser as SChorA. However, they are standalone applications. The last perspective is to integrate the extensions of our implemented tools as a verification plugin for the choreography Eclipse editors, *e.g.*, BPMN 2.0 or WS-CDL.



SChorA Syntax

A SChorA script consists of two parts: declaration and command. In the declaration part, global models of choreographies, local models of roles or services, and composition of local models are declared. The command part describes commands to manipulate the models above, *e.g.*, **conformance** A B to check whether A conforms to B.

Listing A.1 represents the structure of a SChorA script. Literal characters are given in quotes. Vertical bar | separate alternatives. $(X)^+$ represents repetitions $n > 0$ times of X , and $(X)?$ represents repetitions zero or one time of X .

Listing A.1: Structure of SChorA Script

```

1  "DECLARATIONS"
2    (component)+
3  "COMMANDS"
4    (command)+

```

A.1 Declaration Part

The models to analyze are declared in the declaration part which is started by **DECLARATIONS** keyword. Each model is declared inside **component** and **end component** keywords. Listing A.2 represents syntax which is used to declare global and local model.

A model can be described by our choreography algebra language, see Section 3.1,

called *chorD*, which is an extension of *chor* [Qiu et al., 2007] to support data. It can be also given by an STG. In such a case, the STG description has to respect the DOT¹ language.

Listing A.2: Syntax of Model Descriptions Used in SchorA Script

```

1 component: chorD | STG
2
3 chorD: "component" ID "chorD" chor "end component"
4
5 chor : chor ";" chor           //sequential composition
6       | chor "|" chor          //parallel
7       | chor "+" chor          //choice
8       | chor ">" chor           //interruption
9       | "["guard"]" ">" chor    //if
10      | "["guard"]" "*" chor    //loop
11      | "(" chor ")"
12      | event
13
14
15 STG : "component" ID "STG" (state)+ (transition)+ "end component"
16
17 state: INT ";"
18       | INT "[" label=\" ID "\];"
19
20 transition: INT "->" INT " ["label=\" event "\];"
21            | INT "->" INT " ["label=\"[" guard "]" event "\];"
22
23
24 guard: "true" | "false"
25       | expr op expr
26
27 expr : INT | ID
28
29 op : ">" | ">=" | "<" | "<=" | "=" | "!="
30
31 event: "skip"                                //do nothing
32       | ID "[" ID "," ID "]" (dir)?          //event without data
33       | ID "[" ID "," ID "]" dir ID          //event with free variable
34       | ID "[" ID "," ID "]" dir "<" ID ">" //event with bound variable
35
36 dir: "." | "?" | "!"
37
38 ID : ([ "a"- "z", "A"- "Z", "_" ])+ ([ "A"- "Z" ] | [ "a"- "z" ] | [ "0"- "9" ] | "_")*
39
40 INT: ([ "0"- "9" ])+

```

A.2 Command Part

The commands used in a SchorA script are put in the command part which is started by **COMMANDS** keyword. The commands respect the syntax in the Listing A.3.

¹<http://www.graphviz.org/content/dot-language>

Listing A.3: Syntax of Commands Used in SchorA Script

```

1 command: "showTime"
2         | "showStat" (ID)+
3         | "showSTG" (ID)+
4         | "showReachableSTG" (ID)+
5         | "showRealizableSTG" (ID)+ comm
6         | "projection" (ID)+ comm
7         | "conformance" ID ID
8
9 comm   : "SYNC" | "ASYNC_SENDER" | "ASYNC_RECEIVER" | "ASYNC_DISJOINT"

```

The signification of each command is described as follows:

1. **showTime** shows the current time of system. It is useful to calculate the execution time of one or several commands, *e.g.*, put one **showTime** command before and another one just after the block of commands to be calculated then take offset of their results. This command has no parameters.
2. **showSTG** shows graphically STGs
 - Parameters: A list of component names separating by space
 - Example: **showSTG spec impl** will display graphically two STGs corresponding to **spec** and **impl** components.
3. **showStat** displays statistics (number of states, transitions and messages) of STGs corresponding to the specifications in parameters.
 - Parameters: A list of names separating by space
 - Example: **showStat spec impl** will display statistics of STGs corresponding to **spec** and **impl** components.
4. **showReachableSTG** displays reachable STGs corresponding to the specifications in parameters
 - Parameters: A list of names separating by space
 - Example: **showReachableSTG spec** will displays the reachable STG corresponding to **spec** component.
5. **showRealizableSTG** displays realizable STGs corresponding the specifications in parameters
 - Parameters: A list of global component names separating by space, and a communication mode which is one of the following: **SYNC**, **ASYNC_SENDER**, **ASYNC_RECEIVER** or **ASYNC_DISJOINT**

- Example: `showRealizableSTG spec SYNC` displays the realizable STG corresponding to `spec` component under synchronous communication mode.
6. **projection** generates local STGs of global component specifications. The parameters of **projection** command are same as the one of **showRealizableSTG** command.
 7. **conformance** verifies SBBC relation between two components
 - Parameters: a name of implementation component, and a name of choreography component
 - Example: `conformance impl spec` will check whether `impl` component conform to `spec` component

An example of a SChorA script is as in Listing A.4, in which a single line comment is put after `//`, and a multi-line comment is put inside `/*` and `*/`:

Listing A.4: Example of SChorA Script

```

1  /* This script show STG of online shopping choreography which involves 2
   roles: b (buyer), v (vendor)
2   We declare 2 components using DOT and chorD language. */
3  DECLARATIONS
4    component spec STG
5      0; 1; 2; 3; //list of states
6      //list of transitions
7      0 -> 1 [label="request[b,v].<x>"];
8      1 -> 2 [label="x<=0 error[v,b]."];
9      2 -> 1 [label="request[b,v].<x>"];
10     1 -> 3 [label="x>0 response[v,b].<x1>"];
11   end component
12
13   component impl chorD
14     request[b,v].<y> ; (response[v,b].<y1>
15       + [y<=0] * (error[v,b] ; request[b,v].<y>))
16   end component
17
18 //List of commands
19 COMMANDS
20   showSTG shopping impl //display graphically
21   projection spec SYNC //project spec under sync. communication mode
22   conformance impl spec //check conformance

```


Bibliography

- Andrés, C., Emilia Cambronero, M., and Núñez, M. (2010). Passive Testing of Web Services. In *WS-FM*. Springer.
- Andrés, C., Merayo, M. G., and Núñez, M. (2009). Applying Formal Passive Testing to Study Temporal Properties of the Stream Control Transmission Protocol. In *SEFM*.
- Arbab, F. (1998). What Do You Mean, Coordination? In *NVTI*, pages 1–18.
- Arbab, F. (2004). Reo : A Channel-based Coordination Model for Component Composition. *Mathematical Structures in Computer Science*, 14(03):329–366.
- Autili, M., Di Ruscio, D., Di Salle, A., Inverardi, P., Tivoli, M., Ruscio, D. D., and Salle, A. D. (2013). A Model-based Synthesis Process for Choreography Realizability Enforcement. In *FASE*, volume 257178.
- Barker, A., Walton, C. D., and Robertson, D. (2009). Choreographing Web Services. *IEEE Transactions on Services Computing*, 2(2):152–166.
- Basu, S. and Bultan, T. (2011). Choreography Conformance via Synchronizability. In *WWW*, pages 795–804. ACM.
- Basu, S., Bultan, T., and Ouederni, M. (2012). Deciding Choreography Realizability. In *POPL*.
- Bayse, E., Cavalli, A. R., Núñez, M., and Zaïdi, F. (2005). A Passive Testing Approach based on Invariants: Application to the WAP. *Computer Networks*, 48(2):235–245.
- Bentakouk, L., Poizat, P., and Zaïdi, F. (2009). A Formal Framework for Service Orchestration Testing based on Symbolic Transition Systems (long version). In *TESTCOM*, pages 1–19.
- Bentakouk, L., Poizat, P., and Zaïdi, F. (2011). Checking The Behavioral Conformance of Web Services with Symbolic Testing and an SMT Solver. In *TAP*.

- Bergstra, J. A., Ponse, A., and Smolka, S. A., editors (2001). *Handbook of Process Algebra*. Elsevier.
- Bertolino, A., Inverardi, P., Pelliccione, P., and Tivoli, M. (2009). Automatic Synthesis of Behavior Protocols for Composable Web Services. In *Proc. of ESEC/FSE*.
- Bhattacharjee, A. and Shyamasundar, R. (2008). ScriptOrc: A Specification Language for Web Service Choreography. In *APSEC*, pages 1089–1096.
- Bozkurt, M., Harman, M., and Hassoun, Y. (2010). Testing Web Services: a Survey. Technical report, King’s College London.
- Bravetti, M. and Zavattaro, G. (2007). Towards a Unifying Theory for Choreography Conformance and Contract Compliance. In *SC*.
- Bultan, T., Ferguson, C., and Fu, X. (2009). A Tool for Choreography Analysis Using Collaboration Diagrams. In *International Conference on Web Services*, pages 856–863.
- Bultan, T. and Fu, X. (2008). Specification of Realizable Service Conversations using Collaboration Diagrams. In *SOCA*, volume 2, pages 27–39.
- Bultan, T., Fu, X., and Su, J. (2006). Analyzing Conversations of Web Services. *IEEE Internet Computing*, 10(1):18–25.
- Busi, N., Gorrieri, R., Guidi, C., Lucchi, R., and Zavattaro, G. (2006). Choreography and Orchestration Conformance for System Design. In *COORDINATION*.
- Cámara, J., Salaün, G., Canal, C., and Ouederni, M. (2012). Interactive Specification and Verification of Behavioral Adaptation Contracts. *Information & Software Technology*, 54(7):701–723.
- Capizzi, S., Solmi, R., and Zavattaro, G. (2004). From Endogenous to Exogenous Coordination Using Aspect-Oriented Programming. In *COORDINATION*, pages 105–118.
- Decker, G., Kopp, O., and Barros, A. (2008). An Introduction to Service Choreographies. *Information Technology*, 50(2):122–127.
- Decker, G. and Weske, M. (2011). Interaction-centric modeling of process choreographies. *Information Systems*, 36(2):292–312.
- Deng, W. and Lin, H. (2005). Extended Symbolic Transition Graphs with Assignment. *COMPSAC*.

- Diaz, G. and Rodriguez, I. (2009). Automatically Deriving Choreography-Conforming Systems of Services. In *SCC*, pages 9–16.
- Fernandes, F., Passama, R., and Royer, J.-C. (2007). Components with Symbolic Transition Systems: a Java Implementation of Rendezvous. In *CAP*, number i.
- Fernandes, F. and Royer, J.-c. (2007). The STSLib Project : Towards a Formal Component Model Based on STS. In *FACS*.
- Foster, H., Uchitel, S., Magee, J., and Kramer, J. (2005). Tool support for model-based engineering of Web service compositions. *International Conference on Web Services*, 1:95–102.
- Fu, X., Bultan, T., and Su, J. (2004). Conversation Protocols: A Formalism for Specification and Verification of Reactive Electronic Services. *Theoretical Computer Science*, 328(1-2):19–37.
- Gaston, C., Gall, P. L., and Rapin, N. (2006). Symbolic Execution Techniques for Test Purpose Definition. In *TESTCOM*.
- Glenford J. Myers (2004). *The art of software testing*. John Wiley and Sons.
- Hallé, S. and Villemare, R. (2009). Runtime Monitoring of Web Service Choreographies using Streaming XML. In *SAC*.
- Hennessy, M. and Lin, H. (1995). Symbolic Bisimulations. *Theoretical Computer Science*, 138(2):353–389.
- Hwang, S.-Y., Liao, W.-P., and Lee, C.-H. (2010). Web Services Selection in Support of Reliable Web Service Choreography. *ICWS*, pages 115–122.
- Jane Radatz, C. (1990). IEEE standard glossary of software engineering terminology. *IEEE Std 610121990*.
- Jungmayr, S. (2004). Improving testability of object oriented systems. *Naturwissenschaften*.
- Kazhamiakin, R. and Pistore, M. (2006a). Analysis of Realizability Conditions for Web Service Choreographies. *FORTE*, pages 61–76.
- Kazhamiakin, R. and Pistore, M. (2006b). Choreography Conformance Analysis : Asynchronous Communications and Information Alignment. In *WS-FM*.
- Kazhamiakin, R., Pistore, M., and Santuari, L. (2006). Analysis of Communication Models in Web Service. In *WWW*.

- Knuplesch, D., Pryss, R., and Reichert, M. (2012). Data-aware Interaction in Distributed and Collaborative Workflows: Modeling, semantics, correctness. In *CollaborateCom*, pages 223–232.
- Kopp, O. and Leymann, F. (2009). Do We Need Internal Behavior in Choreography Models? In *1st Central European Workshop on Services and their Composition (ZEUS)*.
- Kopp, O., Leymann, F., and Wu, F. (2010). Mapping Interconnection Choreography Models to Interaction Choreography Models. In *Central-European Workshop on Services and their Composition, ZEUS*, pages 1–8.
- Ladani, B. T., Alcalde, B., Cavalli, A., and Landi, B. T. (2005). Passive Testing - A Constrained Invariant Checking Approach. In *TESTCOM*.
- Lanese, I., Guidi, C., Montesi, F., and Zavattaro, G. (2008). Bridging the Gap between Interaction- and Process-Oriented Choreographies. In *SEFM*, pages 323–332.
- Laycock, G. T. (1993). *The Theory and Practice of Specification based Software Testing*. PhD thesis, Sheffield University.
- Lee, D., Netravali, A., Sabnani, K., and Sugla, B. (1997). Passive testing and application to network manager. *Conference on Network*.
- Li, J., He, J., Zhu, H., and Pu, G. (2007a). Modeling and Verifying Web Services Choreography Using Process Algebra. *SEW*, pages 256–268.
- Li, J., Zhu, H., and Pu, G. (2007b). Conformance Validation between Choreography and Orchestration. In *TASE*.
- Li, S., Wang, J., Dong, W., and Zhi-Chang Qi (2004). Property-Oriented Testing of Real-Time Systems. *APSEC*.
- Li, Z. and Chen, H. (1999). Computing Strong/Weak Bisimulation Equivalences and Observation Congruence for Value-Passing Processes. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*.
- Lin, H. (1996). Symbolic Transition Graph with Assignment. In *CONCUR'96*.
- Lohmann, N. and Wolf, K. (2010). Realizability is controllability. In *WS-FM*, pages 110–127.
- Mateescu, R., Poizat, P., and Salaün, G. (2012). Adaptation of Service Protocols Using Process Algebra and On-the-Fly Reduction Techniques. *IEEE Transactions on Software Engineering*, 38(4):755–777.

- McIlvenna, S., Dumas, M., and Wynn, M. T. (2009). Synthesis of Orchestrators from Service Choreographies. In *APCCM*, pages 129–138.
- Mei, L., Chan, W. K., and Tse, T. H. (2009). Data Flow Testing of Service Choreography. In *ESEC/FSE*.
- Mendling, J. and Hafner, M. (2008). From WS-CDL choreography to BPEL process orchestration. *Journal of Enterprise Information Management*, 21(5):525–542.
- Meng, S. and Arbab, F. (2007). Web Services Choreography and Orchestration in Reo and Constraint Automata. In *SAC*.
- Milner, R. (1999). *Communicating and mobile systems: the pi-calculus*, volume 13. Cambridge university press CambridgeUK.
- Moser, O., Rosenberg, F., and Dustdar, S. (2008a). Non-intrusive monitoring and service adaptation for WS-BPEL. In *WWW*.
- Moser, O., Rosenberg, F., and Dustdar, S. (2008b). VieDAME - flexible and robust BPEL processes through monitoring and adaptation. In *ICSE*, page 917.
- Nguyen, H. N., Poizat, P., and Zaïdi, F. (2012a). A Symbolic Framework for the Conformance Checking of Value-Passing Choreographies. In *International Conference on Service Oriented Computing*, pages 525–532.
- Nguyen, H. N., Poizat, P., and Zaïdi, F. (2012b). Online Verification of Value-Passing Choreographies through Property-Oriented Passive Testing. In *IEEE International Symposium on High Assurance Systems Engineering*, pages 106–113.
- Nguyen, H. N., Poizat, P., and Zaïdi, F. (2012c). Passive Conformance Testing of Service Choreographies. In *SAC*, pages 1927–1934.
- Nguyen, H. N., Poizat, P., and Zaïdi, F. (2013). Automatic Skeleton Generation for Data-Aware Service Choreographies. In *ISSRE*.
- Paci, F., Ouzzani, M., and Mecella, M. (2008). Verification of access control requirements in web services choreography. In *SCC*, pages 5–12.
- Pathak, J., Basu, S., Lutz, R., and Honavar, V. (2008). MoSCoE: An Approach for Composing Web Services through Iterative Reformulation of Functional Specification. *International Journal on Artificial Intelligence Tools*, 17(01):109.
- Pavel, S., Noyé, J., Poizat, P., and Royer, J.-C. (2005). A Java Implementation of a Component Model with Explicit Symbolic Protocols. In *SC*.

- Peltz, C. (2003). Web Services Orchestration and Choreography. *Computer*, pages 46–52.
- Poizat, P. (2011). *Formal Model-Based Approaches for the Development of Composite Systems*. Habilitation thesis, Université Paris Sud.
- Poizat, P. and Salaün, G. (2012). Checking the Realizability of BPMN 2.0 Choreographies. In *SAC*, number 1, pages 1927–1934.
- Qiu, Z., Zhao, X., Cai, C., and Yang, H. (2007). Towards The Theoretical Foundation of Choreography. In *WWW*.
- Quinton, S., Ben-hafaiedh, I., and Graf, S. (2009). From Orchestration to Choreography: Memoryless and Distributed Orchestrators. In *FLACOS*. Citeseer.
- Rafiq, O. and Cacciari, L. (2003). Coordination Algorithm for Distributed Testing. *The Journal of Supercomputing*, 24:203–211.
- Repasi, T. (2009). Software testing - State of the art and current research challenges. *2009 5th International Symposium on Applied Computational Intelligence and Informatics*, pages 47–50.
- Roohi, N. and Salaün, G. (2011). Realizability and Dynamic Reconfiguration of Chor Specifications. *Informatica*, 35:39–49.
- Salaün, G., Bultan, T., and Roohi, N. (2012). Realizability of Choreographies using Process Algebra Encodings. *IEEE Transaction on Services Computing*, 5(3):290–304.
- Salaün, G. and Roohi, N. (2009). On Realizability and Dynamic Reconfiguration of Choreographies. In *WASELF*, volume 3.
- Simmonds, J., Gan, Y., Chechik, M., Nejati, S., O’Farrell, B., Litani, E., and Waterhouse, J. (2009). Runtime Monitoring of Web Service Conversations. *IEEE Transactions on Services Computing*, 2(3):223–244.
- Su, J., Bultan, T., Fu, X., and Zhao, X. (2007). Towards a Theory of Web Service Choreographies. In *WS-FM*.
- Sun, J., Liu, Y., Dong, J. S., Pu, G., and Tan, T. H. (2010). Model-Based Methods for Linking Web Service Choreography and Orchestration. In *APSEC*, pages 166–175.
- ter Beek, M., Bucchiarone, A., and Gnesi, S. (2007). Formal Methods for Service Composition. *Annals of Mathematics, Computing & Teleinformatics*, 1(5):1–10.

- Valero, V., Emilia Cambroner, M., Díaz, G., and Macià, H. (2009). A Petri net approach for the design and analysis of Web Services Choreographies. *Journal of Logic and Algebraic Programming*, 78(5):359–380.
- van der Aalst, W., Dumas, M., Ouyang, C., Rozinat, A., and Verbeek, H. (2006). Choreography conformance checking: an approach based on BPEL and Petri Nets. In *ePrints Archive*, pages 1–71. Citeseer.
- Van Glabbeek, R. J. and Weijland, W. P. (1996). Branching Time and Abstraction in Bisimulation Semantics. *Journal of the ACM*, 43(3):555–600.
- World Wide Web Consortium (2005). Web Services Choreography Description Language Version 1.0.
- Wu, G., Wei, J., and Huang, T. (2008). Flexible Pattern Monitoring for WS-BPEL through Stateful Aspect Extension. In *ICWS*, pages 577–584.
- Xiangpeng, Z., Hongli, Y., Zongyan, Q., Yang, H., Zhao, X., Qiu, Z., Pu, G., and Wang, S. S. (2006). A Formal Model for Web Service Choreography Description Language (WS-CDL). In *International Conference on Web Services*, number 605730081, pages 273–287.
- Yoon, Y., Ye, C., and Jacobsen, H. (2011). A Distributed Framework for Reliable and Efficient Service Choreographies. In *WWW*, pages 785–794.
- Zaha, J., Dumas, M., Hofstede, A., Barros, A., and Decker, G. (2006). Service Interaction Modeling : Bridging Global and Local Views. In *EDOC*, pages 45–55.
- Zaïdi, F., Bayse, E., and Cavalli, A. (2009). Network Protocol Interoperability Testing based on contextual Signatures and Passive Testing. In *SAC*.
- Zhou, L., Ping, J., Xiao, H., Wang, Z., Pu, G., and Ding, Z. (2010). Automatically Testing Web Services Choreography with Assertions. In *International Conference on Formal Engineering Methods*.