

Tilburg University

'Correction of unrealizable service choreographies'

Mancioppi, M.

Publication date:
2015

Document Version
Publisher's PDF, also known as Version of record

[Link to publication in Tilburg University Research Portal](#)

Citation for published version (APA):

Mancioppi, M. (2015). 'Correction of unrealizable service choreographies'. CentER, Center for Economic Research.

General rights

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal

Take down policy

If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Correction of Unrealizable Service Choreographies

Proefschrift ter verkrijging van de graad van doctor
aan Tilburg University op gezag van de rector magnificus,

prof. dr. E.H.L. Aarts,

in het openbaar te verdedigen ten overstaan van een
door het college voor promoties aangewezen commissie

in de aula van de Universiteit

op dinsdag 8 september 2015 om 16:15 uur

door

Michele Manciacchi

geboren op 12 september 1981 te Verona, Italië

Promotores:

Prof.dr.ir. M.P. Papazoglou

Prof.dr. W.J.A.M. van den Heuvel

Overige leden van de promotiecommissie:

Prof.dr.ir. M. Aiello

Prof.dr. M.F. Carro Liñares

Prof.dr. D. Grigori

*To my wife,
with all my love
and admiration*

Preface

Pretty much every preface of any doctoral dissertation I have ever read starts by pointing out that a Ph.D. is a long and winding road. Reaching its end is as much a group effort involving mentors, loved ones, friends and other Ph.D. students, as it is the achievement of the individual that will get the title. This preface is no different. And the reason for this orthodoxy is a compelling one: it is an absolutely, incontrovertible fact that achieving a Ph.D. something one can achieve only with the caring help, patient support and steadfast fortitude of a network of a great deal many other people, some of whom are every bit as entitled to the achievement as the author.

First of all, my gratitude goes to my supervisors, Prof.dr.ir. Mike P. Papazoglou and Prof.dr. Willem-Jan van den Heuvel, who found the amazing patience to keep up for years with such an unruly and overly-opinionated Ph.D. candidate. They are also responsible for my involvement in the S-Cube Network of Excellence, where I had the luck to meet Prof.dr. Manuel Carro Liñares, a kindred spirit with whom it has been an immense pleasure to work. Prof.dr.ir. Marco Aiello has been accompanying and mentoring me throughout my entire academic career: first as supervisor to both my bachelor and master theses in Italy, all the way to my Ph.D. defense. Many thanks also go to Prof.dr. Daniela Grigori, who kindly accepted to be part of my committee and who gave many interesting comments to this manuscript. A special mention of honor goes to Alice Kloosterhuis, who was always there for me whenever I needed help at Tilburg University. Deep gratitude goes to Prof.dr. Frank Leymann, who welcomed me to Stuttgart to continue my research and for the many interesting pointers and ideas that made it into this work.

Miriam Adeney wrote: “You will never be completely at home again, because part of your heart always will be elsewhere. That is the price you pay for the richness of loving and knowing people in more than one place.” This says it pretty much all with respect to the past eight years of my life. Friends, colleagues, accomplices, comrades: the Lunch Group was and is all that and more. You have been a second, very large and extremely colorful family to me. (Now the second generation, a.k.a., the “LG natives,” is coming about: beware.)

Words simply fail me to express my gratitude to my parents and sister, who have always been there for me with unquestioning love and unyielding support.

The S-Cube Network of Excellence gave me much more than the chance to work with smart and interesting people. Through it, I met Olha, the amazing woman that became my wife. Insofar this Ph.D. dissertation is concerned, she is above all else the reason why I made it to the end. Granted it, it was not easy: suffice to say I brought her as far as to deserve my very own, home-grown version of Euromaidan to get me to stop procrastinating and write the last, big, scary chapter. (Pots were banged with gusto. Slogans were sang loudly and proudly. It was just *glorious*.)

One last thing is due: a sincere apology to the reader. I allowed myself some liberties. Specifically, this work is peppered some rather atypical footnotes and comments. They deviate from the accepted scientific style, but I could not help myself.¹ May the exacerbated reader take comfort in knowing that there used to be much, much, much more such silliness in the drafts.

¹I personally ~~blame them on~~ credit them to the late sir Terry Pratchett.

Contents

1	Introduction	1
1.1	Motivation	2
1.2	Research Goal	4
1.3	Research Scope	7
1.4	Research Questions	9
1.5	Research Approach	9
1.6	Thesis Structure & Contributions	10
2	Background & Related Work	13
2.1	Choreography Fundamentals	13
2.1.1	Terminology	14
2.1.2	Lifecycle	14
2.1.3	Modeling Paradigms	18
2.2	Interaction Modeling Languages	20
2.2.1	Academic Languages	20
2.2.2	Industrial Languages	24
2.3	Realizability of Choreographies	34
2.3.1	Categorization of Realizability Definitions	34
2.3.2	Categorization of Realizability Analysis Methods	39
2.3.3	Survey of Constructive and Mixed Realizability Analysis Methods	40
2.3.4	Survey of Constraintive Realizability Analysis Methods	43
2.4	Realizability-Driven Evolution of Choreographies	46
2.4.1	Categorization Criteria	47
2.4.2	Survey	48
3	ChorTex	53
3.1	Design Assumptions	53
3.2	Outlook and Syntax	56
3.3	Formal Foundation	59
3.3.1	Basic Definitions	59
3.3.2	Operational Semantics	62
3.4	From ChorTex to Control Flow Graphs	69
3.4.1	Construction Rules	69
3.4.2	Reconciling Actions and Events in Enactments	74
3.5	Well-Formedness Requirements	76
3.6	Differences between ChorTex and Chor	81
3.6.1	Opaque versus Internal Activities	81
3.6.2	Choreography Nesting versus Referencing	82
3.6.3	Finalization Handlers	82

4	Awareness-Based Realizability Analysis	83
4.1	Strong Realizability	84
4.2	Participant Awareness	85
4.3	Annotating Participant Awareness	89
4.3.1	The Awareness Annotation Algorithm	90
4.3.2	Safe Approximation of Input-Values	91
4.3.3	Updating the Output-Values	93
4.3.4	Computational Complexity	95
4.4	Awareness Constraints	96
4.4.1	Observe Your Actions Becoming Enactable	97
4.4.2	Observe Your Actions Becoming Not Enactable	100
4.5	On the Origin of Realizability Defect Types	103
4.5.1	Realizability Defect Types in ChorTex Choreographies	104
4.5.2	From Choreography Modeling Constructs to Participant Awareness Dimensions	104
4.6	Unequivocal Enactments, Revisited	106
4.6.1	Strong Realizability implies Unequivocal Enactments	107
4.6.2	Unequivocal Enactments do not Require Strong Realizability	108
4.7	Discussion	108
4.7.1	Overall Upper-Bound Computational Complexity	109
4.7.2	A Case Against Activity-Terminating Exception Handling (and on the Design of Choreography Modeling Languages)	112
5	Realizability-Driven Evolution	115
5.1	A Change Algebra for ChorTex	115
5.1.1	Insertion Change Operators	116
5.1.2	Update Change Operators	120
5.1.3	Deletion Change Operators	124
5.1.4	Combining Change Operators	126
5.2	Remediation Strategies & Plans	127
5.2.1	Desiderata for Remediation Plans	127
5.2.2	Outlook of Remediation Strategies	128
5.3	On Introducing new Realizability Defects	129
5.3.1	Type 1 Realizability Defects	131
5.3.2	Type 2 Realizability Defects	141
5.3.3	Type 3 Realizability Defects	144
5.4	Remediation Strategies	151
5.4.1	Type 1 Realizability Defects	151
5.4.2	Type 2 Realizability Defects	156
5.4.3	Type 3 Realizability Defects	159
5.4.4	Meta-Strategy: Delete the Affected Node	161
5.5	On Dealing with Multiple Realizability Defects	161
5.5.1	Panta Rei	162
5.5.2	Remediate Early, Remediate Often	163
6	Implementation	165
6.1	Editing ChorTex Choreographies	165
6.2	Visualizing Awareness Models and Realizability Defects	167
6.3	Remediating Realizability Defects	167
6.4	Performance Evaluation	168
7	Conclusions & Future Work	173
7.1	Revisiting the Research Questions	174
7.2	Revisiting the Contributions	175
7.3	Future Work	180

CONTENTS

A	Correctness for Realizability Analysis	185
A.1	Building Peers	185
A.2	Sufficiency of Conditions	186
A.2.1	Stuckness Freedom	187
A.2.2	Language Equivalence	187
A.3	Necessity of Conditions	187
A.3.1	Well-Behavedness	188
A.3.2	Satisfaction of Awareness Conditions	188

Chapter 1

Introduction

It has been said that “the best laid schemes of mice and men often go awry.”¹ Despite the best efforts poured into the planning, sometimes failure is due to unforeseeable contingencies. Some other times, failure is the unavoidable outcome of schemes that were not particularly well laid out to begin with.

In the field of enterprise computing, much of what happens follows deliberately engineered plans encoded in models that specify how the various components of distributed systems must interact with one another. Problems often arise due to contingencies like hardware and networking failures. Sometimes, however, the interactions that the systems have to carry out are designed in such a way that *they simply cannot happen correctly*. Imagine, for example, a system that is meant to react to some situation but that, due to oversights in its design or implementation, is unable to detect the occurrence of that situation.² One could be quick to blame such oversights on shoddy work by designers and developers. However, the truth is that designing and implementing distributed systems is *hard*, and in some fields practitioners simply do not receive adequate support from their tools.

One of the fields that is chronically underserved by modeling tools is *choreography modeling*, the latest incarnation of concurrent system modeling in the scope of Service-Oriented Architecture (SOA) and Business Process Management (BPM). Choreographies are specifications of distributed, message-based behaviors that are carried out by multiple software components or executable processes. The absence of central coordination is the defining feature of choreographies because it is key to achieving the magnitude of scalability necessary in nowadays enterprise systems. Modeling concurrent systems is known to be an arduous and error-prone task; choreography modeling is unfortunately no exception. Indeed, it is disarmingly straightforward to specify choreographies that simply cannot be executed correctly by their participants because of insufficient means of synchronization among them. Such flawed choreographies are said to be *not realizable*. Realizability is a fundamental property for a choreography that is inextricably intertwined with the distributed nature of choreographies [191, 139]. A more precise definition of realizability is postponed until Section 1.2; informally, however, a choreography is realizable if and only if it specifies a distributed messaging behavior that the participants can accurately execute by relying exclusively on the message exchanges therein specified as means of synchronization.

Without realizability, choreographies fail the very task they set out to achieve: the specification of message-based distributed systems that require no centralized coordination. But the intricacies of modeling distributed systems make it hard for choreography modelers to ensure the realizability of choreographies at design time. Despite the large amount of research efforts that have been devoted to realizability of choreographies, the state of the art is rather sparse with respect to

¹From “To a Mouse, on Turning Her Up in Her Nest with the Plough,” a Scots poem written by Robert Burns in 1785.

²“Uninterruptible Power Supply” springs immediately to mind on account of many, horrific stories of *epic* datacenter failure; and do not let the irony of that name be lost on you.

supporting choreography modelers in the hard task of designing realizable choreographies. This situation jeopardizes the adoption of choreographies in the practice and prevents the reaping of the scalability benefits that choreographies promise to bring to enterprise computing. In recognition of this challenge and its relevance, *this thesis is devoted to providing practitioners with effective means of detecting and correcting design defects in choreographies that prevent their realizability.*

This introductory chapter is structured as follows. In Section 1.1 we motivate this research by presenting an example of a choreography that, while rather simple and apparently well-designed, is actually affected by pernicious design defects that prevent its correct implementation. The definition of the research goals and scope driving this work are presented in Section 1.2 and Section 1.3, respectively. The research questions at the basis of this work are outlined in Section 1.4, while the research method is presented in Section 1.5. The chapter ends with Section 1.6, which outlines the contributions provided by this work and the structure of the remainder.

1.1 Motivation

Consider the simplified ordering process modeled as the Business Process Model and Notation (BPMN v2.0) choreography presented in Figure 1.1. This choreography specifies three *roles*: Buyer, Seller and Payment Processor. Roles are personas for parties that will actually execute the choreography. The choreography is initiated by Buyer with the dispatching of the Order message to Seller. Upon reception of the Order message, Seller communicates the Payment Info to Payment Processor to initiate the payment process. At this point in the execution of the choreography there are two, *mutually-exclusive* paths. The first option is that Buyer sends an Order Cancellation to Seller, which then triggers the dispatch by Seller of Payment Process Cancellation to Payment Processor. In turn, this last message exchange ends the choreography. In alternative to Buyer dispatching the Order Cancellation, Payment Processor may send a Payment Solicitation to Buyer, who will answer with the Payment Authorization; after the receipt of the Payment Authorization, Payment Processor concludes the choreography by notifying the completion of the payment to Seller with the Payment Confirmation message.

Albeit very simple, this choreography has one crippling design defect. The choice that governs which of the two alternative paths will be performed, i.e., either finalizing or canceling the order, is harmfully under-specified: it is neither known *who* is in charge of performing the decision – and thus, by implication, every party must perform the decision on its own – nor *which decision criterion* is to be used. Therefore, parties *may not agree* on which of the two, alternative paths is to be followed. This, in turn, causes a race-condition between the cancellation of the order and the dispatch of the payment solicitation. In fact, after the payment processing has been initiated, Payment Processor has only the two following options:

1. Send the Payment Solicitation message at the risk of soliciting the payment of an order that has already been cancelled.³
2. Never send out the Payment Solicitation, thus causing the deadlock of any execution of choreography in which the Buyer does not want to cancel the order.⁴

From the point of view of Payment Processor, there is no way to be sure that its actions will not violate the choreography. This lose-lose situation for Payment Processor is due to a design defect of the choreography that is known as *non-local choice*, i.e., that “[w]hen several processes independently decide to initiate behavior, they might start executing different [alternative ones]” [155]. Non-local choice issues are just one type of a broader range of *realizability defects*, that is, modeling defects that prevent choreographies from being realizable.

³This type of behavior, repeated enough times, would lead to doomsday-level bad PR, if not outright criminal charges for delivering unsolicited payment requests.

⁴Moreover, this way Payment Processor would likely alienate both Buyer and Seller as business partners, as well as setting off its own speedy descent into bankruptcy and causing potentially substantial damage to Seller’s bottom line.

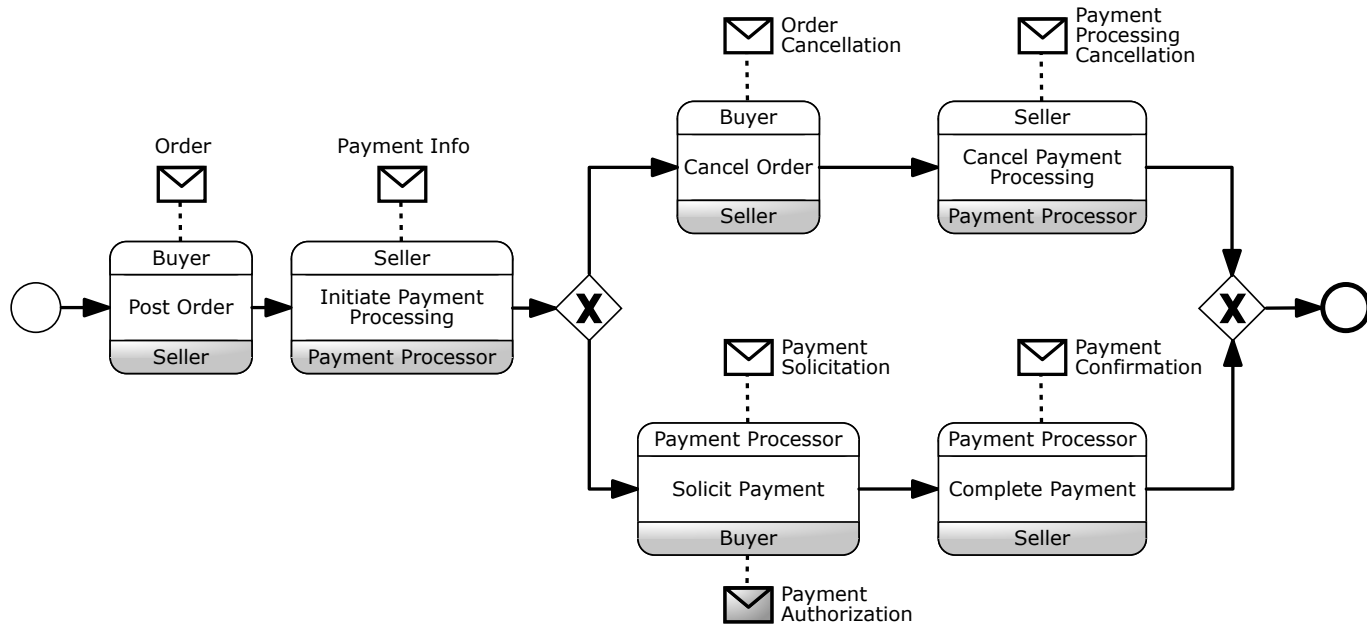


Figure 1.1: A simplified ordering process modeled as a BPMN v2.0 choreography.

The complexity and subtlety of realizability defects demands a broader range of strategies to deal with realizability defects than those currently available. Moreover, these *remediation strategies* must be “at the fingertips” of the choreography modelers to empower them to solve the realizability defects as early as possible and avoid wasting considerable resources to implement, test, debug and root-cause analyze flawed choreographies. In order to understand what type of remediation strategies could be offered to choreography modelers, consider Figure 1.1, which presents two alternative solutions that solve the non-local choice problem under the assumption that the participant playing the role of Buyer knows *which* payment processor is being used. (This is not an unrealistic assumption: usually the payment processor is decided by the buyer among alternatives provided by the seller, and which payment processor is selected is specified in the order.)

The first solution, presented in Figure 1.2a, consists of adding to the choreography shown in Figure 1.1 a message-exchange between the decision and the dispatch of *Payment Solicitation*. Buyer is the participant that sends the newly introduced message *Payment Solicitation Request* and this makes it the only participant that needs to act upon the decision. Therefore, the non-local choice issue is resolved: now the decision is local to Buyer.

However, simply adding message exchanges is by far not the only possible solution to the non-local choice defect affecting the example choreography. For example, Figure 1.2b shows a choreography resulting from the correction of the non-local choice defect by modifying the *Solicit Payment* message exchange into the *Authorize Payment*. Since Buyer is the participant sending the *Authorize Payment* as well as the *Order Cancellation*, the decision has been made *local* to Buyer and the realizability defect is corrected without adding message exchanges to the choreography.

The two proposed solutions to the non-local choice problem represent different trade-offs. On the one hand, every additional message exchange is likely to extend the amount of time that it takes to enact a choreography and thus impact some of its Quality of Service (QoS) aspects. On the other hand, applying the second solution, i.e., the “re-purposing” of the *Solicit Payment* message into the *Authorize Payment*, cannot be applied blindly: the underpinning assumption that the Buyer knows which *Payment Processor* to send the *Authorize Payment* message to must be vetted and validated by the choreography modeler on the basis of information that is not directly represented in the choreography but that instead pertains to the requirements that the choreography must satisfy.

There are two “take-home lessons” that the running example brings across:

1. The choreography modelers require a variety of options to correct realizability defects;
2. The choice of which solution to apply for solving a realizability defect lays ultimately on the choreography modeler.

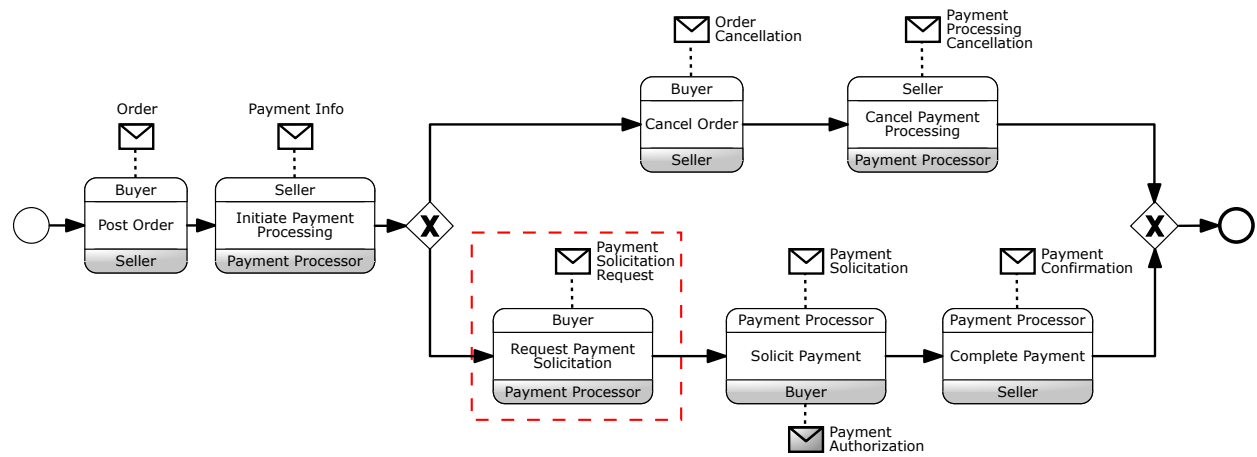
1.2 Research Goal: Remediation of Realizability Defects in Choreographies

The problem of verifying the realizability of specifications has been researched since the late '80s (see, e.g., [3, 167]). Intuitively, a specification is realizable if it “can be implemented by physically possible systems” [3]. In the scope of choreographies, the concept of realizability has been further refined, for example, as follows:

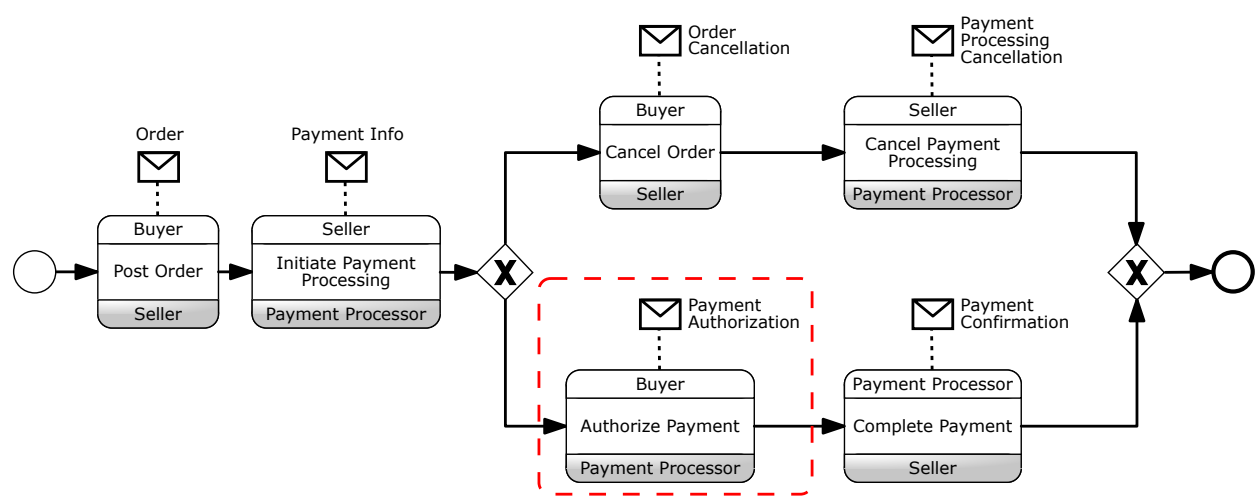
“[...] the possibility to automatically extract from the choreography the behavioral skeletons of the participants so that the concrete implementations, built on the basis of these skeletons, are guaranteed to satisfy the choreography specification.” [116]

The above definition of realizability entails the following series of steps to check whether a choreography is realizable:

1. Project the behavioral skeletons of the participants, which we refer to as *peers* in the remainder of this work;
2. Compose the peers;



(a) Addition of a message-exchange.



(b) Modification an existing message-exchange.

Figure 1.2: Alternative solutions for the realizability defect afflicting the choreography in Figure 1.1.

3. Compare the composition of the peers with the original choreography; if the former is “similar” enough to the latter in terms of messaging behavior, the choreography is realizable.

These steps outline the structure of a prototypical *realizability analysis method*, that is, a process to verify the realizability of a choreography. Of course, other definitions of realizability are possible and not all realizability analysis methods – including the one presented in this work – follow the steps outlined above.

From the perspective of the modeling process of choreographies, realizability defects bear a striking resemblance with programming bugs. In particular, realizability defects are akin to concurrency bugs in multi-threaded programs, which are (in)famous for how hard it is to detect and correct them. Nowadays, software developers can rely on automated, static (i.e., development-time) verification tools to detect many types of bugs like FindBugs⁵ and ThreadSafe⁶ for Java or JSLint⁷ for Javascript. Integrated Development Environments (IDEs) such as Eclipse⁸ or IntelliJ IDEA⁹ integrate such static verification tools to perform continuous verification *while* the code is being developed. However, spotting a bug is not necessarily enough to facilitate and streamline its correction by the developer: the IDE must also provide a quick, semi or fully automated way of correcting those errors, e.g., removing the problematic statement, changing the visibility of a variable or importing missing packages and classes. We argue that the same must be available for choreography modeling as well.

Figure 1.3 presents our vision, inspired by the similarities between software bugs and realizability defects, to support modelers in the modeling of realizable choreographies. At modeling-time, whenever a change is applied to the choreography, the IDE verifies “in the background” its realiz-

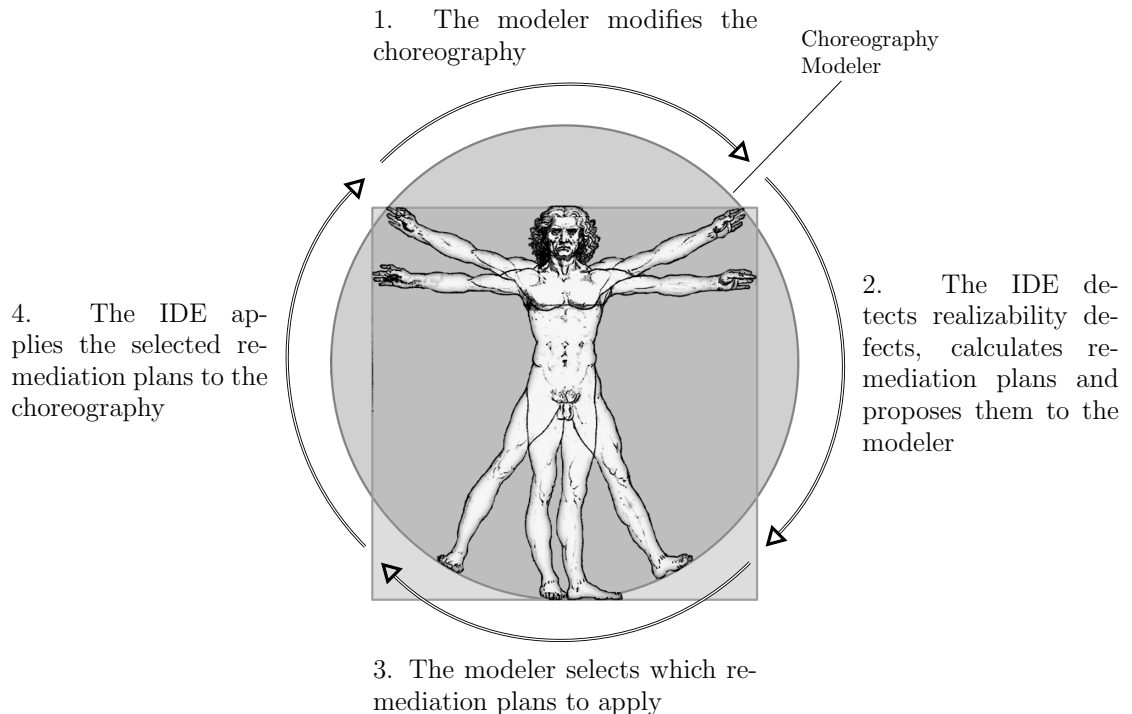


Figure 1.3: Detection and correction of realizability defects at modeling-time from the point of view of the modeler.

⁵FindBugs website: <http://findbugs.sourceforge.net>

⁶ThreadSafe website: <http://www.contemplateltd.com>

⁷JSLint website: <http://www.jshint.com>

⁸Eclipse website: <http://www.eclipse.org>

⁹IntelliJ IDEA website: <http://www.jetbrains.com/idea>

ability. If the realizability analysis detects one or more realizability defects, the IDE automatically calculates the possible remediation plans. The choreography modeler then selects which remediation plans are applied by the IDE to the choreography. Of course, the choreography modeler may elect not to apply any of the remediation plans (not shown in the picture), for example when she is aware of the current realizability defect and has already a set of changes in mind to correct it later in the modeling.

Embracing this vision, this thesis aims at accomplishing the following **goal**:

To design and implement algorithms to automatically detect at modeling-time the realizability defects that afflict a choreography and devise proposal for changes, called remediation plans, that solve them.

1.3 Research Scope

From the previous discussion it is evident that modelers need better support for detecting, understanding and correcting realizability defects at modeling time, before any enactment of the choreography is ever performed. This leads us to the following **problem definition**:

How to analyze the realizability of choreographies and present to the modeler diagnostic information and remediation plans to solve the detected realizability defects?

The state of the art of choreography modeling is extremely broad and varied, and not all choreography modeling languages suffer from the the same pitfalls in terms of realizability. In fact, the types of realizability defects that may occur depend on the constructs that are provided by the specific choreography modeling language. Solving the problem of realizability defects in a general fashion is outside the scope of any single PhD Thesis. To satisfactorily achieving our goal, we restrict this research to a feasible scope by means of the following assumptions:

Interaction Choreographies: In this work we focus on interaction choreographies, i.e., those choreographies that specify the messaging behavior from a global perspective that focuses on the ordering of message exchanges among the roles. The interaction paradigm comes natural to modelers, as it closely resembles workflows as well as service orchestrations modeled with languages like Business Process Execution Language for Web Services (WS-BPEL) and, to some extent, even imperative programming languages.

Ordering of Message Exchanges: In the example provided in Section 1.1, the realizability defect is the result of a modeling flaw in the order of the message exchanges that can be performed by the participants. In those choreography modeling languages that, like BPMN v2.0 Choreography Diagrams, explicitly model messages as data-structures (e.g., using XML Schema Definition (XSD)) and that allow data-based decisions, realizability is also concerned with the content of messages being exchanged and how the participants perform decisions based on them.

Considering the “data dimension” of choreography modeling languages complicates considerably how realizability is verified. Depending on the expressiveness of the data-structures used and the queries performed on them, it has been shown that it may even render realizability undecidable [157]. Therefore, in this thesis we will avoid the data dimension of choreographies and focus on realizability defects that result exclusively from flaws in the order of the message exchanges specified by choreographies.

Messaging Technology Agnosticism: Another aspect of choreographies that we are not going to consider is the *grounding* to any particular implementation technology for the participants. For example, we are not going to make any assumption about what is the language used to actually model the peers and implement the participants, may it be an orchestration language like WS-BPEL or a general-purpose programming language like Java. We are also not going to make any

direct assumption about the communication protocol used by the participants to communicate with each other, e.g., Simple Object Access Protocol (SOAP), Java Message Service (JMS) or Hypertext Transfer Protocol (HTTP).

Asynchronous Communication Model: While we are not going to assume any specific communication protocol, we do need to assume a particular communication model (which is a necessary aspect of any definition of realizability, see Section 2.3.1.2). According to the taxonomy of communication models provided in [132], we are assuming the following type of *asynchronous messaging*:

- sending strategy: non-blocking sending, always succeed
- buffers:
 - ordering: ordered
 - capacity: unbounded
 - quantity: multiple buffers (one per participant)

Functional Aspects of Remediation Plans: While all remediation plans should *functionally* solve the remediation defects, their impact on non-functional aspects of the choreography can be very different. As glue among information system of different individuals, enterprises and organizations, choreographies often involve participants that, while in business with each other, have strict, well-motivated policies and restrictions concerning confidentiality of data. For example, under no circumstances is acceptable to have one’s complete medical record sent by the insurance company to his employer just because sharing that one message prevents a realizability defect from occurring in the choreography.

Another non-functional aspect of choreographies that may be negatively affected by remediation changes are connected to QoS aspects. Intuitively, the more message exchanges are introduced among participants, the longer it will take to execute the choreography (e.g., due to networking latency) and the more bandwidth will be consumed.

While the selection of which remediation plan to be applied is a task that pertains exclusively to the modeler, she should ideally be supported in the decision with figures estimating the impact of each available remediation plan on the overall choreography. However, the evaluation of non-functional impact of changes to choreographies is very much “terra incognita” in the state of the art, and would very possibly warrant several PhD Theses just by itself. Therefore, we will leave as future work the evaluation and prioritization of remediation plans based on non-functional aspects, see Section 7.3.

“Green Field” Choreographies: If a choreography has already been implemented by the participants, changes to it in terms of the specified messaging behavior require the participants to adapt some or all of their implementations. Changes that have this type of disruptive effect are known as *deep changes* [14]. While deep changes are considered something to be avoided whenever possible, they are simply inevitable when trying to fix the realizability of choreographies that have already been implemented. Since disruption is unavoidable, one should try to minimize as much as possible the impact on existing participant implementations. Similarly to the case of QoS impact of remediation plans, the effect of deep changes on existing participant implementations and the according prioritization is an enormous challenge, which we necessarily must to leave as future work. Therefore, we assume a “green field” scenario, where the choreography has not yet been implemented and were the remediation of realizability defects does not have to take into account the impact on existing participant implementations, see Section 7.3.

1.4 Research Questions

The problem statement enunciated in Section 1.3 has been broken down in the following research questions:

What is the state of the art of modeling choreographies, realizability analysis and correction of realizability defects? Which modeling languages and formalisms are available for choreographies? What are the salient characteristics of a definition of choreography realizability? Which methods are used to verify the realizability of choreographies? Which remediation strategies have already been proposed and how can they be categorized?

What is an adequate definition of choreography realizability? Given the usage of choreographies in the state of the art, what characteristics should a definition of choreography realizability have?

Given the chosen definition of realizability, how can it be verified in an effective manner? How are the constructs of a choreography modeling language related to the realizability defects that may occur in its choreographies?

Which remediation strategies can be defined to correct realizability defects? What alternatives can be offered to the modeler in terms of remediation changes? When multiple realizability defects are found in a choreography, in which order should they be tackled?

How can the theoretical results be leveraged to provide an integrated approach to solving realizability defects during the modeling process of choreographies? What are the strengths and limitations of the proposed solution? A usable implementation of the proposed solution must be implemented in a way that is immediately usable by choreography modelers.

1.5 Research Approach

The research presented in this work has been carried out according to the following process:

Review and analysis of the state of the art: In order to better understand the problems connected with choreographies, it was necessary to investigate in depth the extremely rich and diverse state of the art, which spans across industry standards, industrial research and academic research. A large number of publications relevant to the topic of this work have been found. Analyzing their strengths and weaknesses has greatly helped in both shaping our findings as well as avoiding duplicated work.

Problem definition: At the beginning of a research path such as a PhD, the goal is to understand the domain of the research and formulate short-term research hypotheses. As the research proceeds, the problem statement and the research hypotheses evolve and mature. In the case of the research detailed in this work, the final research questions have been outlined in Section 1.4.

Solution design and prototyping: Despite containing a consistent amount of formalisms and theoretical research, this thesis has an eminently practical goal: give choreography modelers tools to support them in their daily modeling work. Given this very practical goal, the solutions have been designed and prototyped iteratively. This thesis proposed a novel choreography modeling language, called ChorTex, the design of which has grown organically with the realizability analysis method and remediation strategies proposed in this work. As more constructs were added to the choreography modeling language, the realizability analysis method and the remediation strategies

were extended to accommodate them. To make sure that the focus stayed on effectively supporting choreography modelers, the theoretical and formal parts of this research have been carried out in parallel with the implementation of the prototype. More often than not, having an early, tangible implementation of some ideas and techniques has contributed greatly to improving them. For instance, the display of diagnostic information to the choreography modeler has grown organically with the realizability method, and some details of the latter have been determined by the needs of the former (e.g., using Control Flow Graphs to model participant awareness instead of more abstract methods like constraint-driven stageful model-checking).

Validation: According to the categorization of validation approaches for design science proposed in [110], the contributions proposed in this work have been validated in *analytical* and *testing* fashion. Analytical validation has been carried out in the shape of the mathematical proofs of the correctness of algorithms and methods we introduced and the evaluation of their computational complexity (i.e., static analysis) and performance measurements of the prototype (i.e., dynamic analysis). The testing validation has consisted of extensive choreography modeling and correction of realizability defects undertaken by this author with the prototype implemented on the basis of the proposed realizability analysis and remediation strategies.

Evaluation: Finally, the strengths and shortcomings of the proposed solution have been examined in Chapter 7, and the latter are adopted as a baseline to define future research directions.

1.6 Thesis Structure & Contributions

The contributions to be provided by the thesis are the following, broken down according to the respective chapters.

Chapter 2

Background & related work: The chapter aims at providing the knowledge of fundamentals and state of the art of choreographies in order to understand the remainder of the thesis. Specifically, the following topics are covered:

- Choreography fundamentals: terminology, lifecycle of choreographies and outlook of the existing choreography modeling paradigms;
- Choreography modeling languages and formalisms that adopt the interaction modeling paradigm.

Categorization of realizability definitions: Analysis and categorization of the various definitions of realizability is proposed. Our categorization builds on and improves others already found in the state of the art, as well as offering an novel new dimension concerning how the realizability definitions are formulated.

Categorization and survey of realizability analysis methods: An analysis of realizability analysis methods is proposed and correlated to the previous categorization of realizability definitions. The realizability analysis methods found in the state of the art are classified according to the proposed categorization.

Categorization and survey of remediation strategies: We provide a categorization of remediation strategies for unrealizable choreographies that builds on top of the previous categorization of realizability definitions as well as the dimensions of automatism and types of modifications to be applied to choreographies. The remediation strategies found in the state of the art are classified according to the proposed categorization.

Chapter 3

ChorTex: The chapter presents the syntax, operational semantics and design assumptions that underpin ChorTex. ChorTex is a choreography modeling language of our own devising that builds on the work of [205] and that is adapted to model interaction choreographies that rely on asynchronous, one-to-many message exchanges.

In a nutshell, the decision of working with a process algebra like ChorTex is that the block-based structure of process algebras gives ample opportunity to reuse algorithms and methods developed in the field of programming languages such as Control-Flow Analysis. These techniques allow to specify performant realizability analysis methods for ChorTex and to present diagnostic information of the realizability defects as annotated Control Flow Graphs, which are familiar to many developers and modelers with programming backgrounds.

Chapter 4

Realizability analysis: The chapter presents the definition of realizability that we adopt for ChorTex choreographies, namely *strong realizability*, as well as an analysis method for verifying them on ChorTex choreographies. The analysis method is based on the Control Flow Graphs introduced in Chapter 3 and the concept of *participant awareness*, i.e., if a participant can observe during an enactment the execution of an action such a message exchange or a decision. The representation of the control-flow graph of the choreography annotated with the participant awareness and the constraints on the participant awareness that must be satisfied to guarantee strong and halting realizability of the choreography enable the display of visual, comprehensive diagnostic information for choreography modelers to understand the realizability defects and their impact on the behavior of the choreography.

Categorization of realizability defects: Given the constructs of ChorTex, there is a limited amount of types of realizability defects that can afflict ChorTex choreographies. (It is shown in Section 2.4 that it is generally true that the types of realizability defects that can afflict a choreography depend intrinsically on the constructs provided by the adopted choreography modeling language.) The types of realizability defects applicable to ChorTex choreographies are investigated, categorized and related with the language constructs that are involved in their occurrence.

Chapter 5

Change algebra: The thesis provides an exhaustive set of basic change operators that allow to modify every aspect of ChorTex choreographies. The change operators are formally defined alongside constraints that ensure that the change operators preserve the well-formedness of ChorTex choreographies.

Invariants to preserve realizability: A number of invariants are defined to ensure that, upon applying the change operators defined by the change algebra, no new realizability defects are introduced.

Remediation strategies: A variety of remediation strategies are presented that tackle all the types of realizability defects potentially affecting ChorTex choreographies. The remediation plans generated by these remediation strategies are specified on the basis of the previously-defined change algebra for ChorTex choreographies.

Guidelines to deal with multiple realizability defects: The remediation of a realizability defect has often “side-effects” on other realizability defects. Sometimes, a remediation plan may solve multiple realizability defects. Other times, applying one remediation plan may restrict the available options to solve another realizability defect. Based on observations on the realizability

analysis method defined in Chapter 4, guidelines are proposed to ease the task of dealing with multiple realizability defects in a single choreography.

Chapter 6

Implementation: The chapter presents the Eclipse-based IDE for modeling ChorTex choreographies that has been developed to prototype the concepts presented in this thesis. The prototype provides the following features:

1. Modeling of ChorTex choreographies based on Xtext¹⁰, a framework for the development of text-based Domain-Specific Languages (DSLs);
2. Visualization of the control-flow based diagnostic information based on Eclipse Zest¹¹;
3. Implementation of all the proposed remediation strategies and the application of the resulting remediation plans to the choreography being modeled;
4. The possibility for the user to compare the current status of the choreography with the one resulting from the application of a remediation plan using the Eclipse Compare framework¹², which is of immediate understanding to any developer familiar with using versioning systems like Subversion¹³ or Git¹⁴ from within Eclipse.

Chapter 7

Conclusions & future work: The chapter summarizes the findings of this thesis and, in light of them, revisits the problem statement that has been presented in Section 1.3. Additionally, future research directions are proposed, both in terms of extensions of the ChorTex language, and the application of the concepts introduced in this thesis to other Choreography Description Languages.

¹⁰Xtext website: <http://www.eclipse.org/Xtext>

¹¹Eclipse Zest website: <http://www.eclipse.org/gef/zest>

¹²Eclipse Compare framework website: <http://eclipse.org/eclipse/platform-team>

¹³Apache Subversion website: <https://subversion.apache.org/>

¹⁴Git website: <http://git-scm.com/>

Chapter 2

Background and Related Work

The goal of the present chapter is to provide the preliminaries that are necessary for discussing in the remainder of this thesis the realizability analysis and realizability-driven evolution of choreographies. This chapter is organized as follows. Section 2.1 provides a primer on the fundamentals of choreographies. The major choreography modeling languages that are used to model interaction choreographies are presented in Section 2.2. Section 2.3 discusses the notion of realizability, examining the realizability definitions and realizability analysis methods found in the state of the art. Finally, the existing approaches for modeling-time evolution of service choreographies are surveyed in Section 2.4.

2.1 Choreography Fundamentals

Over the past few years, choreographies have attracted a large amount of interest in the research communities surrounding SOA and BPM. For example, a query on Google Scholar¹ for “service choreography” or “service choreographies” returns north of 400 results.² The wide berth of the state of the art is even more impressive considering that there are a number of concepts coming from different communities that are closely related to service choreographies, such as multi-agent protocols and service contracts [48, 2] – in the meaning of “behavioral descriptions of Web Services” [64].

As discussed in [78, 184], the convergence of such diverse research communities has lead to a considerable “blurriness” with respect to what a choreography actually is.³ In [184], the author identifies the following three classes of choreographies:

Process choreographies are modeled using constructs like configurable business transactions (see, e.g., [165]) that are semantically close to those employed in BPM models and that are tailored to the use case of business-to-business integration;⁴

Services choreographies are modeled using constructs specific to Web Services and SOA technologies and are closely related with service orchestrations, i.e., service compositions that “represent control from one party’s perspective” [166];

¹Google Scholar: <http://scholar.google.com>

²Precisely, 404 as of June 10, 2014.

³Indeed, the title “Do we need a refined choreography notion?” of [184] is quite probably one of the most blatant violations ever of Betteridge’s law of headlines, namely that: “Any headline which ends in a question mark can be answered by the word no.” [39]

⁴For the name of this class of choreographies we have borrowed the terminology proposed by [79]; in [184], process choreographies are actually called “Business-to-Business Integration (B2Bi) choreographies,” which sounds unfavorably obscure and antimnemonic.

Conceptual choreographies abstract from specific implementation technologies, are modeled using constructs driven by the purpose of analysis and verification and may be used to complement/analyze the BPM layer as well as the orchestration layer.

In the scope of this thesis, choreographies are defined as follows:

Definition 2.1 (Choreography). A *choreography* is a prescriptive model of the collective, distributed, external messaging behavior occurring among two or more *roles* defined in terms of the ordering of the messages that are to be exchanged.

According to the taxonomy of choreographies proposed in [184], the above definition of choreography is consistent with the “conceptual choreography” one, which reflects the scope for our research outlined in Section 1.3 as avoiding assuming any specific implementation technology.

2.1.1 Choreography Terminology

Figure 2.1 provides an outlook of the terminology adopted in this thesis regarding choreographies. As a prescriptive model of distributed messaging behavior among roles, a choreography intentionally specifies a set (of possibly infinitely many) *conversations*, i.e., sequences of message exchanges performed by the *participant implementations*. The participant implementations are those software components or systems that expose the messaging behavior mandated by the roles assigned to their respective *participants*, which are entities such as organizations or individuals.

Participant implementations may (and usually do) realize behaviors that are not mandated by the choreography, e.g., message exchanges among their own parts or sub-components, or interactions with users or systems not taking part in the choreography; such *internal behaviors*, however, are of no importance from the point of view of the choreography. As detailed later in Section 2.1.2, depending on the tooling at their disposal, participants may be able to obtain from the choreography *peers*⁵, i.e., models that encode the messaging behaviors associated with particular roles. An *enactment* is the collective execution by the participant implementations of one of the conversations specified by the choreography. That is, the participants perform an enactment by having their participant implementations exchange messages with each other.

In the same way programming languages can be classified as functional, object-oriented, imperative and so on, choreographies can be classified according to the *choreography modeling paradigm* they adopt. In a nutshell, choreography modeling paradigms are different styles of modeling choreographies using modeling constructs that accentuate different perspectives, e.g. the global perspective on the overall choreography, the perspective of specific roles or an abstract perspective based on explicitly modeling the evolution of the status of business artifacts, such as the status of a shipment or of a procurement order, that are affected by the choreography. An outlook of the main choreography modeling paradigms is provided in Section 2.1.3.

2.1.2 Choreography Lifecycle

In the practice of SOA and BPM, choreographies are used to achieve the following goals [139]:

Choreography Usage 1 (Specification and communication). A choreography acts as a *technical contract* among the participants, which collaborate with each other towards a shared business goal [79]. As such, the choreography establishes how the participant implementations must interoperate with each other in terms of message exchanges in order to accomplish this shared business goal. Therefore, the role assigned to a participant represents its *obligations* towards the others.

Choreography Usage 2 (Facilitate participant implementations). During the software development process of the participant implementations and depending on the employed choreography modeling language and relative tooling, it is often possible to quickly generate peers from the choreography that act as prototypes or skeletons for the participant implementations.

⁵Also known in the state of the art as “business protocols” [38, 178].

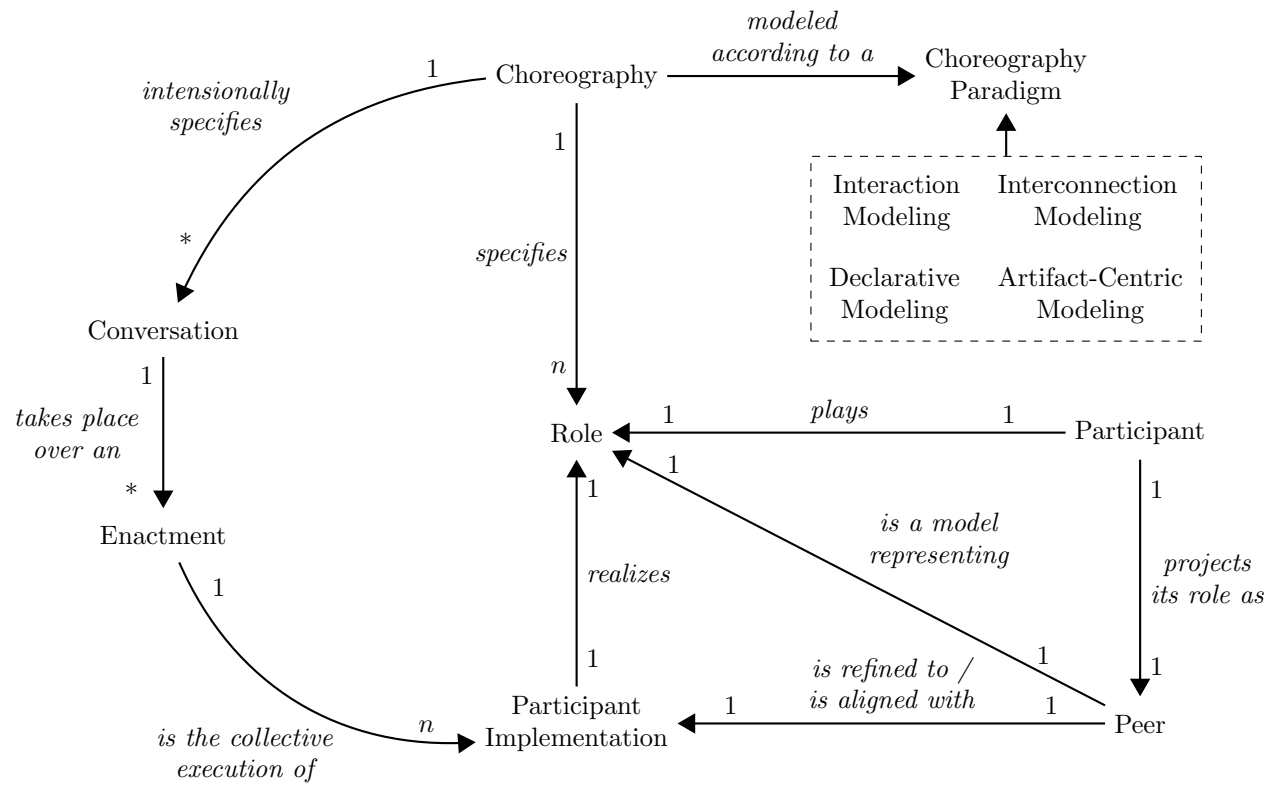


Figure 2.1: A map of key choreography-related concepts.

Choreography Usage 3 (Monitoring the enactments). Monitoring facilities such as those integrated in Enterprise Service Bus (ESB) middleware, may use the choreography to verify that its enactments do not violate it [120], for example for auditing purposes (see, e.g., [63]). Monitoring can either be performed during the enactment (*on-line* monitoring) by observing the message exchanges occurring between participant implementations or after its completion using logs (*off-line* monitoring, also known as *post-mortem* monitoring).

These uses underpin the lifecycle of choreographies and relative participant implementations that is presented in Figure 2.2. (Other, finer grained lifecycles are of course possible, depending, for example, on the software development process utilized to implement the participant implementations.) In its role of specification of message-based interactions among systems provided by business entities, a choreography is likely to be updated over time to reflect evolving requirements. Similarly to the case of software maintenance (see, e.g., [67]), the evolution of a choreography may be motivated by a variety of goals, such as addressing emerging business needs of its participants (see, e.g., [137]), achieving better QoS characteristics or simply accommodate the changes in behavior of single participants or of the communication middleware they employ.

At the reaching of a development milestone, such as the completion of the modeling process or the reaching of a “beta” status, the choreography is made available to the participants to allow them to prepare their participant implementations. Of course, it may be the case that some participants already have participant implementations available and are do not necessarily receive the choreography, which is for example the case of participants that provide public APIs that

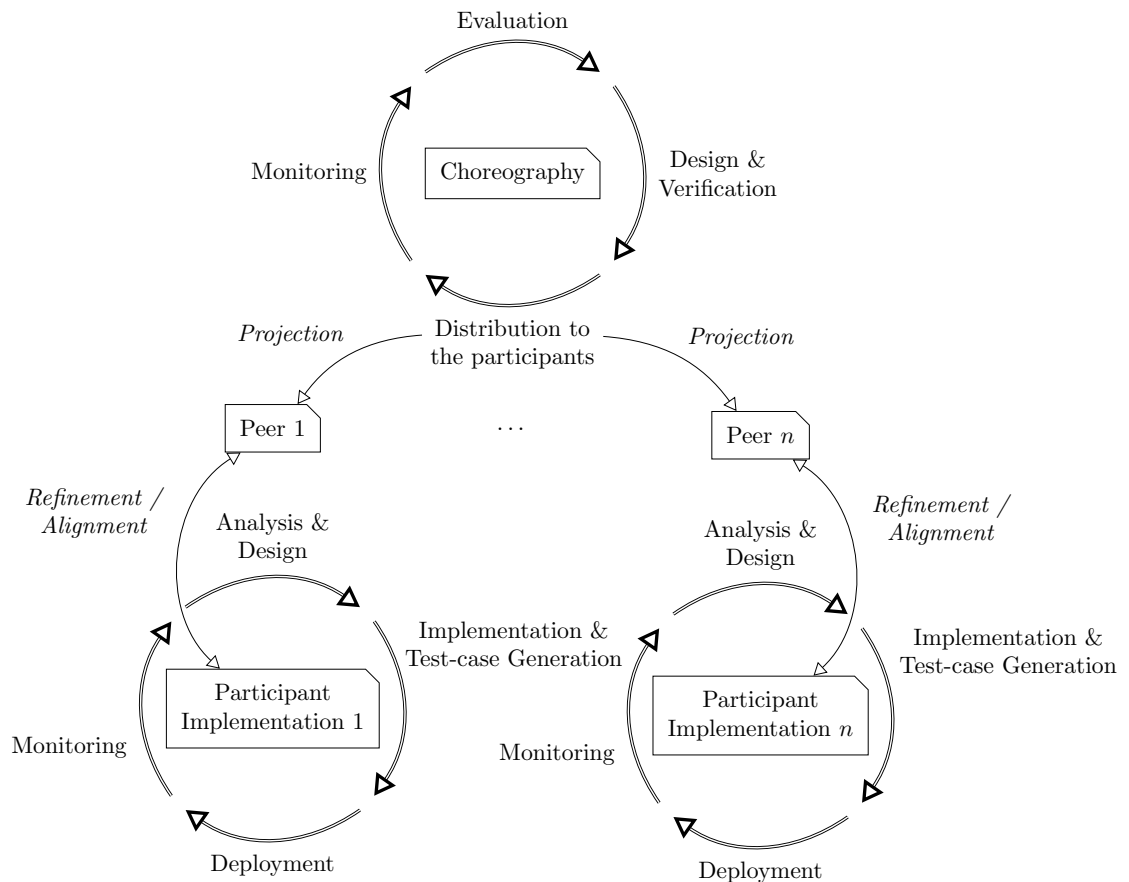


Figure 2.2: Correlations between the lifecycles of a choreography and the participant implementations in SOA (adapted from [80]).

are “integrated” with the other participant implementations by means of the choreography. For instance, when Web APIs such as Google Maps or Search are integrated in some application, the API provider generally does not need to adapt in the least the behavior of its service: the burden of integration is on the consumers of the services.

How the choreography is distributed to the participants is usually related with the goals it aims at accomplishing. Choreographies that represent technical contracts among participants follow the usual, highly-varied ways in which technical contracts are disseminated, e.g., publication on a website, inclusion in specifications submitted to standard bodies á la RosettaNet⁶ or simply sending it over email to the contact persons of the various participants. The research community has produced a few approaches to choreography registries that are reminiscent of service brokers for Web Services descriptions (see, e.g., [16, 113, 10]), but at the best of this author’s knowledge there is neither adoption nor industry-driven efforts towards such technologies.⁷

After the choreography has been distributed, a participant may *project* the messaging behavior mandated by its role as a peer. The algorithms to project a peer from a choreography depend on the modeling languages used to specify the choreography and are usually integrated in the tools used to model the choreography. Since a peer is limited to modeling the point of view of one participant, it does not represent those parts of the distributed messaging behavior that do not involve that peer’s role, and in particular it does not comprise the message exchanges that take place between other roles.

By adding implementation specific details and internal behaviors, a peer is iteratively *refined* into the final participant implementation. While not all participant implementations are realized via refinement of peers, it is nevertheless a very common process when the participant implementation consists of an executable service orchestration. This is the case, for example, of peers modeled as abstract WS-BPEL processes that are refined into executable WS-BPEL processes. The executable processes, once deployed on an orchestration engine, realizes the participant implementation (see, e.g., [122]). Other ways of realizing participant implementations are as many as the software development processes employed by the participants, such as “from scratch” implementations using general-purpose programming languages or modifications to existing software systems.

In case of mismatch between the messaging behavior of the elected participant implementation and the peer, the participant implementation needs to be *aligned*, i.e., modified to satisfy the requirements set by the role. If modifying a participant implementation is not possible (e.g., in the case of integrating publicly available APIs without a direct involvement of their providers in a consortium), the choreography must be modified to accommodate the changes in the messaging behavior (not represented in Figure 2.2 for simplicity). Besides acting as a baseline for participant implementations, peers can be used to generate test cases (see, e.g., [201, 213]) for verifying the *conformance* of the participant implementations, i.e., that the messaging behavior exposed by the participant implementation corresponds to the one mandated by the role. The literature encompasses a wide spectrum of definitions of conformance that hinge on different relations between the messaging behavior of a participant implementation and its the respective role, see, e.g., [25, 117, 49, 22].

The monitoring of a choreography consists of observing its enactments with the goal of collecting data on violations, performance-related QoS attributes, etc. The data collected can be used to correct violations of the choreography during the enactment, e.g., in case a participant implementation is acting differently than its role mandates (see, e.g., [180]), as well as correcting issues in further iterations of the modeling of the choreography. Monitoring may happen also “internally” to single participant implementations for reasons like governance and auditing and may fuel further, “endogenous” iterations of the development of the participant implementations that are not triggered by changes in the choreography.

⁶RosettaNet website: <http://www.rosettanet.org>

⁷As a side note, repositories dedicated to choreographies seem rather “overkill.” It should be pretty a straightforward task to simply adapt existing Universal Description Discovery and Integration (UDDI) registries, should the need for discoverable, meta-data rich choreographies arise in the practice.

2.1.3 Choreography Modeling Paradigms

The goal of this section is to provide an overview of the existing choreography modeling paradigms. The two dominant choreography modeling paradigms in the state of the art are known as *interconnection* (Section 2.1.3.1) and *interaction* (Section 2.1.3.2). Additionally, other two paradigms have recently emerged, namely *declarative* (Section 2.1.3.3) and *artifact-centric* (Section 2.1.3.4).

2.1.3.1 Interconnection Choreography Modeling

In the interconnection modeling paradigm, the roles are each modeled in separation. The messaging behavior of each role is specified in terms of the order of the messages that are dispatched and received by that role. This is the case, for example, of choreographies modeled using BPEL4Chor [83, 85], Web Service Modeling Ontology (WSMO) Choreographies [174, 173] or BPMN v2.0 Collaboration Diagrams; an example of the latter is shown in Figure 2.3a. The peers are then correlated with each through *wiring*, i.e., by means of constructs that connecting elements in different peers. In the case of BPMN v2.0 Collaboration Diagram, the wiring constructs are called *message flows*. Message flows “tie together” the elements in different peers that constitute a message exchange, connecting the elements that generate the message in the sender’s peer with those that receive the message in the recipient’s peer. Since the roles are modeled in relative separation, the projection of the peers from the choreography is as simple as “severing” the wiring among the roles.

2.1.3.2 Interaction Choreography Modeling

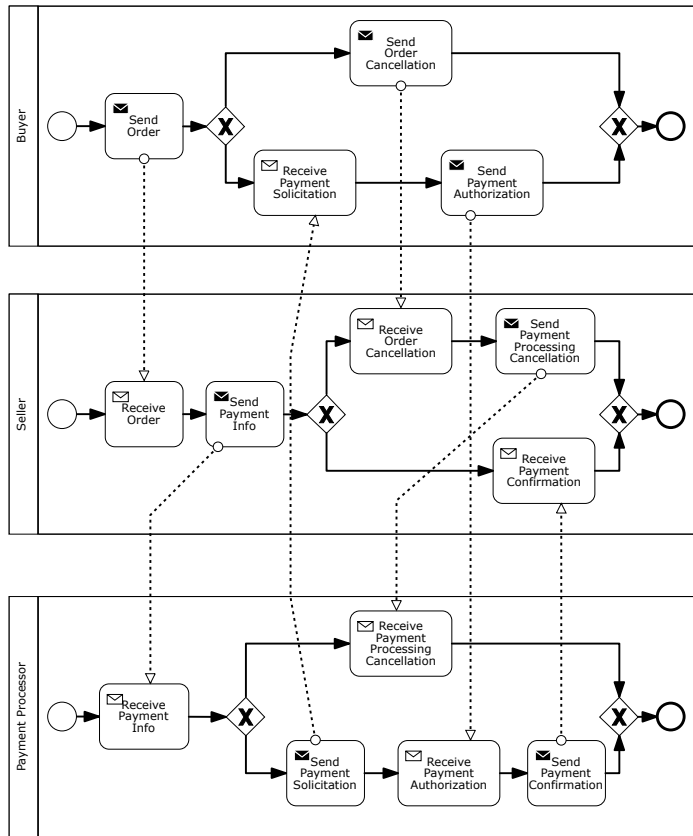
The interaction modeling paradigm foresees the specification of the messaging behavior of choreographies as “atomic interactions [among the roles] that are related through global behavioral constraints” [79]. The roles in interaction choreographies are specified implicitly, meaning that the choreography is modeled from a *global* perspective. In the BPMN v2.0 Choreography Diagram shown in Figure 2.3b, each activity specifies either one or two message exchanges between the participants annotated on the activity itself. An outlook on the large variety of choreography modeling languages adopting the interaction modeling paradigm is provided in Section 2.2.

As a paradigm, interaction modeling has the advantage of allowing the modeler to specify exactly what message exchanges need to be enacted without having to delve in the “internal” details of each participant. The immediacy of interaction modeling, however, comes at the cost of the complexity of projecting the peers: first of all, an interaction choreography may not be realizable, making it impossible to actually produce peers that can enact it correctly. Moreover, since the projected peers have to represent the point of view of one single participant, they are virtually always modeled using a modeling language other than the choreography’s.

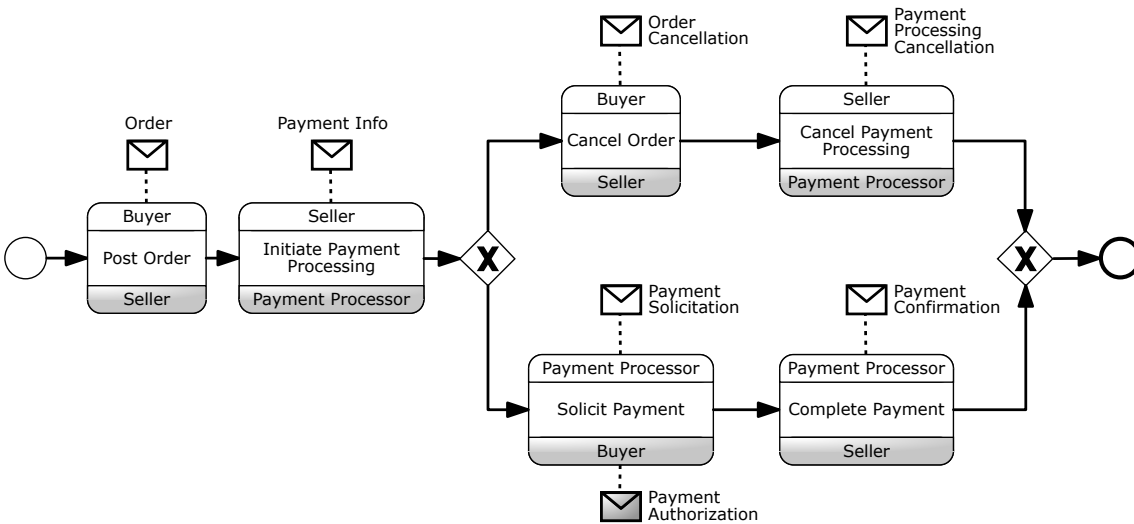
In the scope of service choreographies, the projected peers are specified using modeling languages, like WS-BPEL, that have been designed to model service orchestrations. The discrepancies between orchestration and choreography modeling languages introduce considerable complexity in the projection and in keeping the alignment between projected peers and the choreography when one of them evolves. This complexity is proven by the extremely large amount of research devoted, for example, to reconciling WS-BPEL and Web Services Choreography Description Language (WSDL) (see Section 2.2.2.3). In this author’s opinion, this is very strong (albeit circumstantial) evidence that orchestration and interaction choreography modeling languages have to be designed from the very beginning to be interoperable with each other.

2.1.3.3 Declarative Choreography Modeling

The declarative choreography modeling paradigm is inspired by declarative programming languages like Prolog: the order of the message exchanges among participants is modeled implicitly by means of constraints that define pre- and post-conditions. To the best of this author’s knowledge, declarative choreography modeling is currently being investigated only in the scope of academic research (see, e.g., DecSerFlow [152] and [89, 88]), but no declarative industrial languages are yet available.



(a) BPMN v2.0 Collaboration Diagram.



(b) The BPMN v2.0 Choreography Diagram used as running example in Chapter 1.

Figure 2.3: Two equivalent choreographies modeled according to the interconnection (Figure 2.3a) and interaction (Figure 2.3b) paradigms.

2.1.3.4 Artifact-Centric Choreography Modeling

The artifact-centric paradigm is the declension in the scope of choreographies of the “artifact-centric business process model” approach in BPM (see, e.g., [40, 115, 72]): the conversations that can be enacted are specified implicitly in terms of artifacts, their states, and how the message exchanges occurring among the participant implementations alter the states of the artifacts. In other words, the control flow that orders the message exchanges “emerges” from the way the states of the artifacts evolve [96]. Similarly to the case of declarative choreographies, the industry has not yet embraced the artifact-centric paradigm, which is currently been investigated solely in the scope of academic research [133, 96].

2.2 Interaction-based Modeling Languages for Choreographies

As outlined in Section 1.3, in this work we concentrate on interaction choreographies. In the remainder of this section, the state of the art of interaction choreography languages is divided in *academic languages* (Section 2.2.1) and *industrial languages* (Section 2.2.2).

2.2.1 Academic Languages for Interaction Choreographies

Academic languages are choreography modeling languages that have been proposed by the academic community and virtually all build directly on top of formal frameworks like automata, Petri nets or process calculus. Their formal foundation and the usually compact abstract syntax facilitate the definition of algorithms for processing the choreography models and formal proofs of their properties [79]. Academic choreography modeling languages focus on conceptual choreographies; the lack of grounding with particular technologies and the resulting lack of vendor support and integration in the technological ecosystem strongly limit their adoption in the practice. Another factor that limits the adoption of academic languages by practitioners is their perceived lack of usability: the lack of graphical notation and the often unwelcoming syntax are very often unsurmountable obstacles.

While the direct impact of academic languages on the practice is limited, their usefulness lies in the deeper understanding they enable of fundamental, theoretical and largely open problems like realizability and compatibility analysis. Moreover, useful constructs studied in academic languages tend eventually to find their way in industrial languages. For example, the most prominent industrial choreography modeling languages, namely WS-CDL and BPMN v2.0, display clear influences from formalisms based on π -calculus and Petri nets.

Most academic languages are either based on process algebras (see Section 2.2.1.1) or automata “et similia” (see Section 2.2.1.2). The remainder is built on top of other formalisms that have so far catered a smaller amount of interest (see Section 2.2.1.3).

2.2.1.1 Academic Languages Based on Process Algebras

Among the different families of academic languages to model interaction choreographies, those built on process algebras (also known as process calculi) have received by far the most research scrutiny. This should not come as a surprise: after all, the seminal works in the field of process algebras such as [149, 112] had exactly the goal of enabling the formal specification of distributed messaging systems made of independent agents.⁸ Unfortunately, the fragmentation of the choreography research community has lead to a particularly splintered state of the art insofar process algebras are concerned. Indeed, it is often extremely challenging to understand how two academic languages based on process algebras differ from each other except for matters of notation. Nevertheless, the

⁸The interested reader will find in [20] a compelling presentation of the history of process algebras and in [9] a very interesting discussion on the use of process algebras for the design of Domain-Specific Languages (DSLs).

state of the art of modeling interaction choreographies with process calculi is rather vibrant and promising.

Some works in this area, for example [98, 136], build directly or indirectly on top of the Calculus of Communicating Systems (CCS) [150]. CCS and most of the process calculi derived from it support *channel passing*, i.e., the passing from one participant to another of a “reference” used to communicate with a third participant. In the scope of choreographies, channel passing realizes the service-interaction pattern “Request with referral” [29], which allows participants to dynamically join the enactments at run-time on an “invitation-basis,” instead of being statically determined at modeling time. Naturally, the expressive power of channel passing comes at the cost of more verification to be performed, e.g., to make sure that those participant that are supposed to send a message to a dynamically-defined participant actually have received a reference “pointing” to it beforehand [55, 206]. The extension of CCS-like process algebras to deal with exception throwing and handling is explored in [205] (without support for channel passing) and [58] (with channel passing).

Some approaches, such as [56], build on top of π -calculus [181] (which is itself based on CCS). Among these, Signal Calculus [43, 70, 71] adds the notion of *location* to the frame of π -calculus, giving the opportunity of modeling activities local to certain participants and, through them, specify in an abstract way how the content of the messages is to be calculated. In multi-agent protocols [27], the notion of role in CCS-like calculi is refined from just a channel identifier to a set of methods that are invoked by the other roles over message exchanges; this extension is rather interesting, as it fits well the meta-model of service interfaces underpinning Web Services Description Language (WSDL).

Session types (see, e.g., [192]) is a flavor of process algebra that adopts a type system-like formalization [90] and that has attracted a considerable amount of research interest in the scope of choreographies. Multi-party session types with asynchronous communication are investigated in [60, 62, 69, 61]. The relationship between multi-party session types and projections for the roles specified using communicating finite state machines is investigated in [87]. Similarly to the previously-mentioned works on adding exception-handling to CCS based process calculi, [59] investigates the same based on session types between two participants.

2.2.1.2 Academic Languages Based on Automata and Other State-Transition Systems

Most of the works available in the state of the art about modeling choreographies with Finite State Machines (FSMs) or State-Transition Systems (STSs) adopt the interconnection modeling approach (see Section 2.1.3.1): each role is modeled as a separate automaton and then their composition is verified in terms of, e.g., deadlock-freeness (see, for example, [169]). There are, however, a few approaches that use various types of FSMs and STSs to model interaction choreographies. As shown in Figure 2.4, the FSM or STS states represents states of the enactments of the choreography, while the message exchanges are annotated on the transitions that connect the states.

The differences in the various approaches for interaction modeling are often rather minute and mostly limited to matters of notation; the articles that propose them, however, differ from one another in terms of which problems related to interaction choreographies are analyzed. For example, reduction rules for checking the conformance of peers projected for the roles are investigated in [25]. How different communication models (e.g., synchronous, asynchronous with queues of limited or unlimited size) lead to different definitions of realizability is treated in [116]. Whether a change to a peer requires “on cascade” changes to others is investigated in [171]. In [172], instead, is studied how to “fix” the peers projected from an unrealizable choreography by to them adding message exchanges (which are not added to the choreography itself). This last approach offers a different take to remediating realizability issues than the one investigated in this work: instead of fixing what is broken (the choreography), the error is “worked around” in the implementations of the single peers. Such an approach has very interesting applications in scenarios where promptly modifying the choreography is generally outside the reach of the participants, e.g., in case of

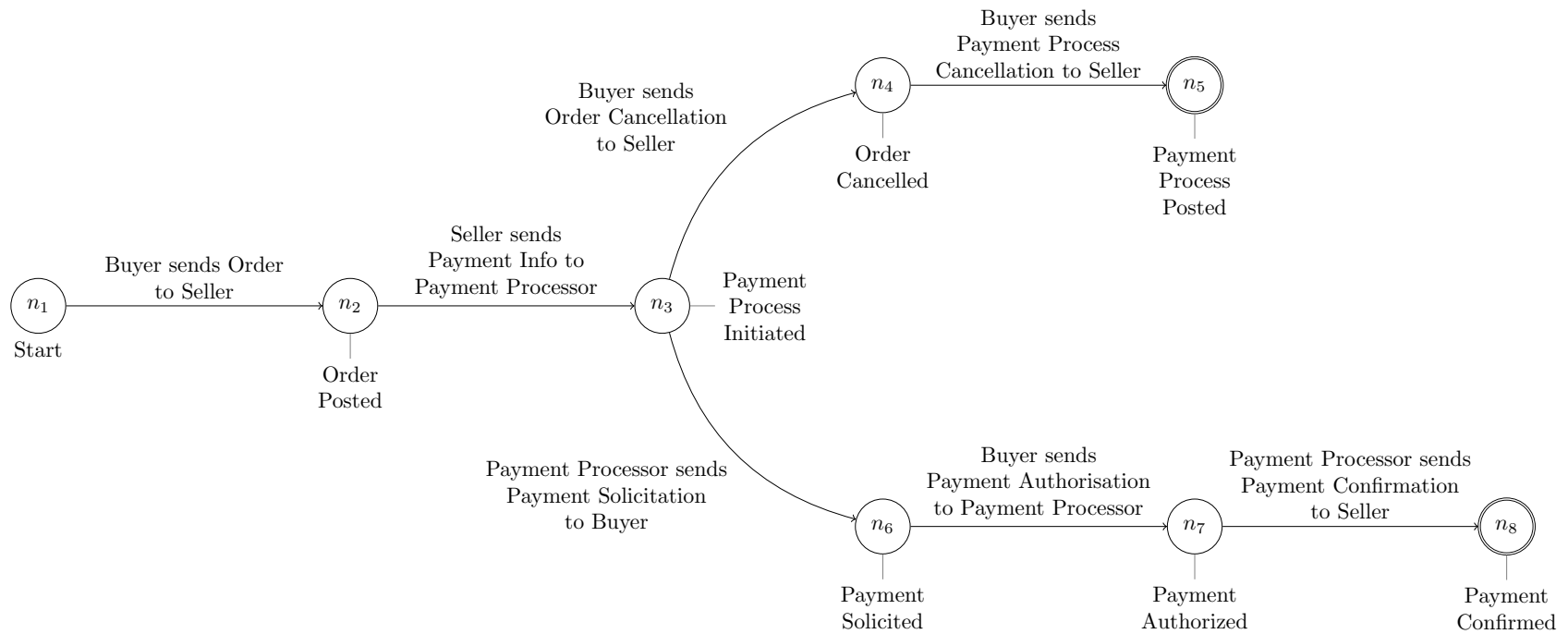


Figure 2.4: A FSM modeling the choreography displayed in Figure 2.11; the message exchanges are annotated on the transitions in the form of “[sender] sends [message] to [recipient]”.

defective specifications mandated by standard bodies.

There are also a few works that propose extensions to the general approach of interaction modeling with automata by allowing the specification of conditions that enable or disable the transitions, and therefore further restrict when the message exchanges can be executed. In [138], this author has proposed a framework based on timed automata-based that allows the specification of time constraints on the transitions. In [102, 100], guarded automata are proposed to model and verify interaction choreographies; the conditions associated with the transitions are based on the content of the messages that are been exchanged.

2.2.1.3 Academic Languages Based on Other Formalisms

Even though most formal research on interaction choreographies builds on top of process algebras, STSs or FSMs, there are a number of academic languages for modeling choreographies that explore the use of different formalisms, namely:

Petri nets are mostly used in conjunction with the interconnection paradigm, examples of which are Operational Guidelines (see, e.g., [190]) and Open Workflow Nets (see, e.g., [182]). Petri nets, however, are used in [81, 86] to model interaction choreographies: the places represent partial states of the enactment, while the transitions represent message exchanges among the roles.

Dynamic Logic is a modal logic that has been applied by the Multi-Agent Systems (MAS) research community to tackle “the problem of reasoning about actions and change” [23]. In [24], the authors apply DyLOG, a programming language based on dynamic logic, to the encoding and verification of interaction choreographies specified as Agent UML (AUML) sequence diagrams [161], an adaptation of Unified Modeling Language version 2.x (UML 2.x) sequence diagrams (see Section 2.2.2.2) for describing message-based interactions among agents.

Distributed States Temporal Logic [154] is a modal logic for reasoning on distributed applications that combines the description of location of computation (where is something calculated) and its temporal aspect (when is something calculated). In [153], the authors present a logic framework based on distributed states temporal logic to model interaction choreographies with constructs inspired by WS-CDL (see Section 2.2.2.3).

SCIFF [8] builds on top of Abductive Logic Programming with the aim of specifying interactions among open societies of agents (i.e., groups of agents that are not necessarily known a-priori). In [7], the authors present a language based on SCIFF to model interaction choreographies and verify during the enactment of the choreography (or after its completion based on the trace of the conversation), whether the enactment conforms to the choreography.

Let’s Dance [210, 82, 211, 84, 209] is a choreography modeling language originated in the research community that, due to its appealing graphical notation and rich expressiveness, may pass for an industrial language (which is not, due to the lack of standardization process or, as far as this author can tell, vendor involvement). The operational semantics of Lets Dance has been formalized by means of π -calculus in [82].

Figure 2.5 presents as an example a Let’s Dance choreography inspired by an e-Government scenario. Each message exchange taking place between two roles is represented by a box. Sender and recipient are marked inside nested boxed at the top-left and top-right corner of the message exchange box, which is split in the middle by an arrow line. The tip of the arrow line points to the recipient of the message exchange.

The ordering among message exchanges is specified by means of control flows connecting them. There are three types of control flows in Let’s Dance. A **precedes** control flow, denoted by a line with open arrow-head, means that the message exchange that is targeted by the arrow-head

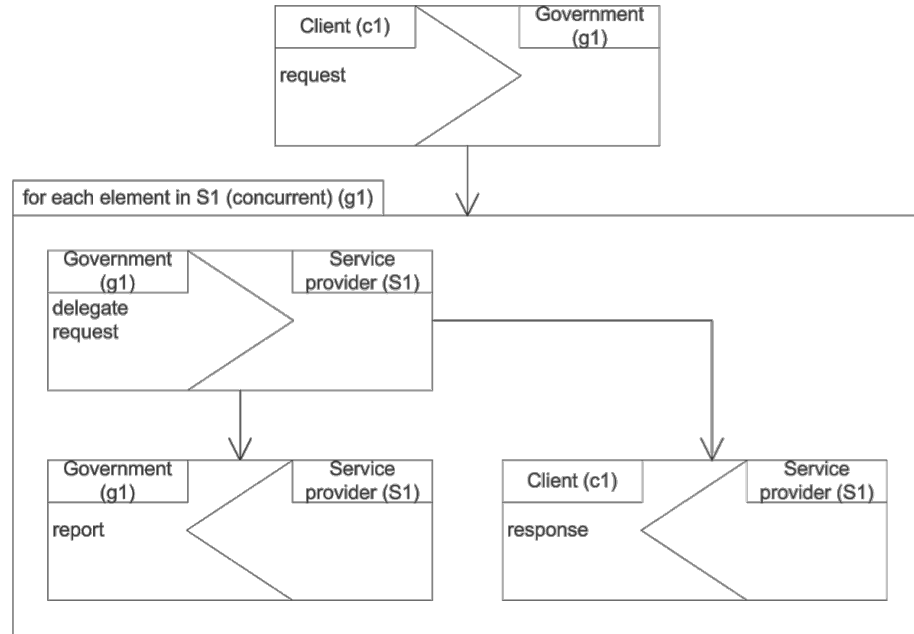


Figure 2.5: An example of Let’s Dance choreography [210].

can take place only after the source of the control flow has. **Inhibits** control flows, denoted by control flow crossed with a perpendicular line (not displayed in the example), mean that the occurrence of the source message exchange prohibits the execution of the target (but not vice-versa). Finally, **weak-precedes** control flows (not displayed in the example) denote that the target message exchange cannot take place until the source one has either been carried out or inhibited.

Multiple message exchanges can be grouped in **complex interactions**, graphically represented by boxes surrounding the composed message exchanges. Additionally, message exchanges and complex interactions can be executed conditionally or iteratively on the basis of expressions that are encoded in labels attached to them, such as the “for each element in ...” label attached to the complex interaction in Figure 2.5.

2.2.2 Industrial Languages for Interaction Choreographies

Modeling languages – not only those for choreographies – are said to be “industrial” if they undergo a standardization process in the scope of organizations or consortia like the World Wide Web Consortium (W3C)⁹, the Object Management Group (OMG)¹⁰ or the Organization for the Advancement of Structured Information Standards (OASIS)¹¹.

While the state of the art of academic choreography modeling languages has been advancing rapidly in the past few years, the same cannot be said for their industrial counterpart. With respect to the state of the art captured in 2006 by [30], the only notable difference is the introduction of BPMN v2.0. This stability is, per se, not necessarily a bad sign: an abundance of new industrial languages would cause fragmentation in the practice and would require large investments in training of the workforce, tooling and integration with the software ecosystem, et cetera. One may even interpret this stability as a sign of maturity, an empirical proof that the needs of the practice of choreography modeling have been satisfactorily met. However, at the best of this author knowledge there is no scientific proof of any such “happy” state of affairs. On the contrary, one of the most significant problems affecting industrial languages for modeling choreographies has remained largely

⁹W3C website: <http://www.w3c.org>

¹⁰OMG website: <http://www.omg.org>

¹¹OASIS website: <https://www.oasis-open.org>

ignored, namely the *lack of formal foundation*. In the scope of SOA and BPM, the standards that define most industrial languages lack formal, normative specifications of the operational semantics of their modeling constructs. In the words of [50], “although often masterpieces of apparent clarity, [the standards] usually suffer from inconsistency, ambiguity and incompleteness.” In the scope of industrial languages for modeling choreographies, the lack of formal foundation constitutes a significant challenge for the understanding and handling of issues like realizability [30].

Another side-effect of the lack of formal foundation of modeling languages for choreographies (and of other service composition approaches) is the noticeable dispersion of efforts by the research community, which has tried to “step in and fill the gap.” Consider, for example, the tens of different formalizations of WS-CDL (see Section 2.2.2.3). Despite the valuable contribution towards exploring how formalism can be applied to industrial languages, such “a posteriori” formalizations have no normative value. Therefore, there is no guarantee that vendor implementations will conform to the formally-expressed behavior; this, in turn, jeopardizes the impact on the practice of research results based on these “a posteriori” formalizations.

Even though the state of the art of industrial languages for modeling interaction choreographies has seen little change in the past few years, it is nevertheless too large to be entirely covered in this chapter. Faced with the need of prioritizing some languages over others, one would favor those languages that have the most impact on the practice, which could be measured in terms of, e.g., adoption rate. Unfortunately, this author is not aware of any figures of the kind. In need for a better criterion, the remainder of this section focuses on those industrial languages for which there is related work in terms of realizability analysis, namely: Message Sequence Charts (MSCs) (Section 2.2.2.1), some types of UML 2.x models (Section 2.2.2.2), WS-CDL (Section 2.2.2.3) and BPMN v2.0 Choreography Diagrams (Section 2.2.2.4). Other industrial languages for interaction modeling that may be of interest to the reader, but that we will not cover here, include Web Service Choreography Interface (WSCCI) [17], Web Service Conversation Language (WSCL) [26], United Nations Centre for Trade Facilitation and Electronic Business (UN/CEFACT) Modeling Methodology (UMM) [114], Electronic Business using eXtensible Markup Language (eXML) Business Process Specification Schema (BPSS) [54], WSMO Choreography [174] and OWL-S [140].

2.2.2.1 Message Sequence Charts

MSC [147] is a standard released in 1992 in the scope of International Telecommunication Union (ITU) and updated multiple times over the years. Figure 2.6 shows an MSC that models the

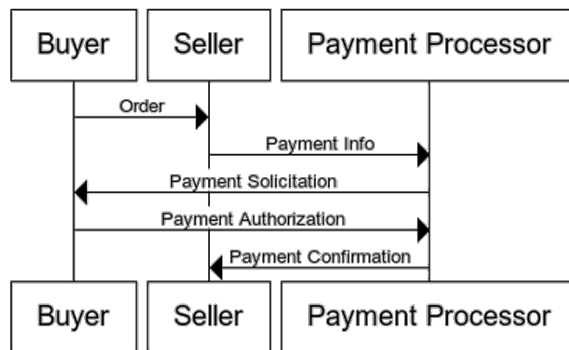


Figure 2.6: An MSC that models the choreography displayed in Figure 2.3a.

conversation of the choreography displayed in Figure 2.3a in which the acquisition is successfully finalized. Each MSC describes one particular conversation; nevertheless, MSCs are considered interaction choreographies because of their capability of modeling roles and their message-based interactions from a global perspective (see e.g. [13, 79]). Each role is represented by a **lifeline**, which is the combination of the box containing the name of the role and the dashed line originating from it. Message exchanges between roles are one-to-one and asynchronous; they are denoted by

message flow originating from the lifeline of the sender and terminating in the lifeline of the receiver. The ordering of message exchanges involving roles is defined based on the order in which the message flow touched the lifeline: messages closer on the lifeline to the box occur earlier than those further from it.

High-level Message Sequence Chartss (HMSCs) have been added to the MSC standard in 1996 as means to combine multiple MSCs using control flow constructs such as iteration, choice and concurrency [141]. Furthermore, HMSCs can be composed recursively, allowing the nesting of HMSCs into others. HMSCs can intentionally define multiple (and possibly infinite) conversations, and in this respect are much more similar to other interaction choreography languages than MSCs.

2.2.2.2 UML 2.x Diagrams

UML 2.x provides three different types of models that have been considered for specifying interaction choreographies: communication diagrams, sequence diagrams and interaction overview diagrams.

Communication diagrams are simplified version of Unified Modeling Language version 1.x (UML 1.x) collaboration diagrams. Communication diagrams, an example of which is shown in Figure 2.7, depict roles as nodes. The roles are connected with each other by lines that denote one-to-one message exchanges between the connected roles. The actual message exchanges are denoted as “labels” attached to the connections. The direction of the arrow of a connection defines which role is sender and which the recipient. The order in which message exchanges take place during enactments is encoded by the *sequence numbers* contained the associated labels. Sequence numbers use the so-called *nesting notation* to specify the correlations between the message exchanges: for example, in Figure 2.7 the message exchange **Order** has sequence number 1, and since it triggers **Payment Info**, the latter has sequence number 1.1. Choices and conditional message exchanges are modeled using the [condition] notation that defines the triggers. Sequence numbers and choices make for rather cumbersome modeling and have been pointed out as the reason why MSCs are far more often used in the practice than communication diagrams [99].

Sequence diagrams are extremely similar to MSCs in terms of both notations and scope (i.e., one conversation); the similarities are obvious when comparing the MSC Figure 2.6 with the equivalent sequence diagram displayed in Figure 2.8. The semantics of sequence diagrams, however, has some fundamental differences with respect to that of MSCs. While MSCs are focused on modeling

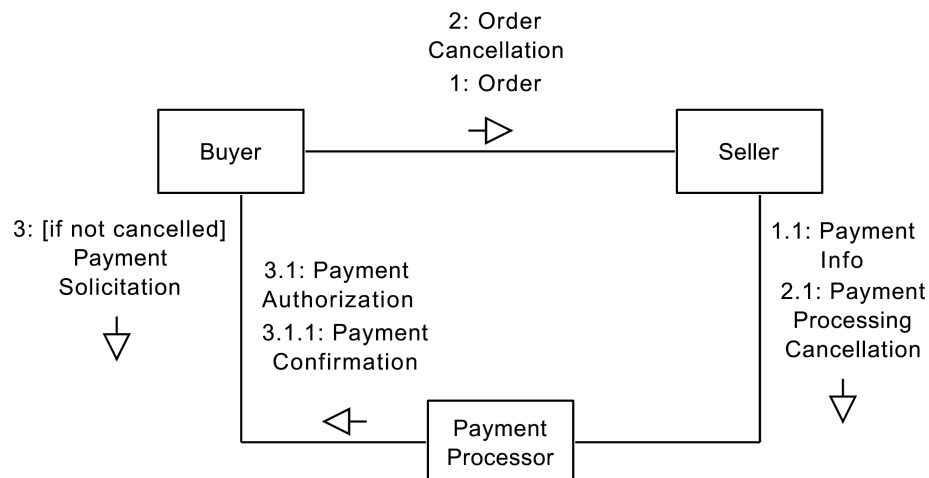


Figure 2.7: A UML 2.x communication diagram that models the choreography displayed in Figure 2.3a.

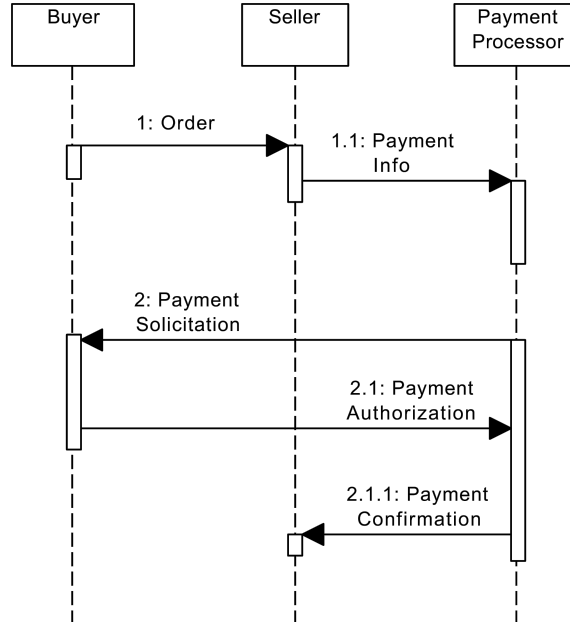


Figure 2.8: A UML 2.x sequence diagram that models the choreography displayed in Figure 2.3a.

autonomously executing entities and their message exchanges, sequence diagrams – like the rest of UML 2.x – have their focus on objects and the invocation of operations on them. Because of the focus on operation invocation, the arrows connecting lifelines in sequence diagrams represent synchronous, blocking invocations, instead of asynchronous message exchanges as in MSCs. Despite the difference in focus, however, sequence diagrams are sometimes used to model interaction choreographies, like in the case of the probably most-quoted publication about choreographies, namely [166].

Interaction overview diagrams combine multiple sequence and communication diagrams by means of control flow constructs and, thus, are the UML 2.x equivalent of HMSCs (see Section 2.2.2.1). Figure 2.9 presents an interaction overview diagram that models the same choreography as displayed in Figure 2.3a. The order of the execution of the nested diagrams is specified using the choice, iteration and other control flow constructs that are found in activity diagrams. As a result, the nested diagrams can be seen as complex message exchanges that go beyond the usual one-way messaging or response-reply message exchange patterns that are common in other choreography modeling languages.

2.2.2.3 Web Services Choreography Description Language

The W3C has published WS-CDL as a candidate recommendation [195] in November 2005. WS-CDL, which supersedes in the scope of W3C its predecessors WSCI and WSCL, is loosely based on π -calculus, borrowing from the latter some key terminology and concepts.

Figure 2.10 shows an example of WS-CDL choreography. The **relationship** construct specifies the “intention to interact between two roles” [176]. Which roles interact over a relationship is defined by the respective **relationshipType**, omitted in the example for reasons of brevity. In WS-CDL, **variables** are used to store the content of the messages exchanged by the participants, the intermediate calculations used to perform decisions and the identifiers of **channels**. A channel represents 1-to-1 communication medium between two roles specified in terms of those roles’ endpoints, messaging and security policies, etc. The order of the message exchanges that take place over the channels is specified using control flow constructs like **sequence**, **parallel** and **choice**

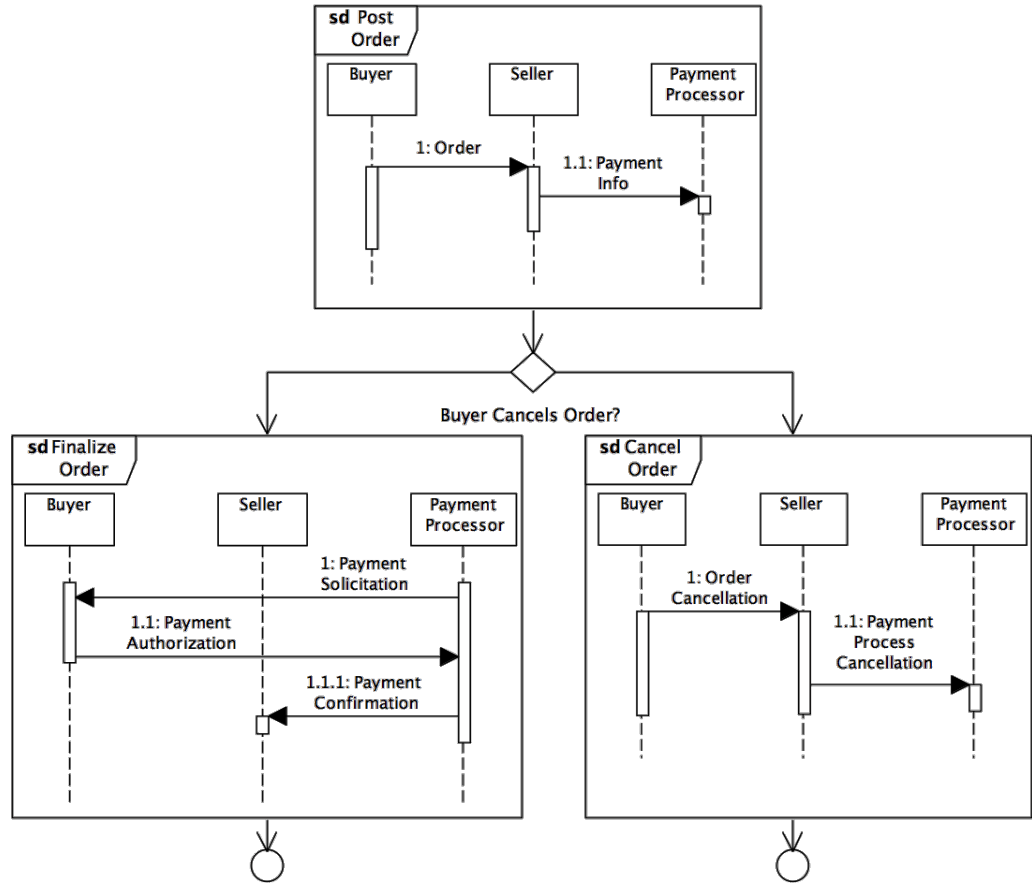


Figure 2.9: A UML 2.x interaction overview diagram that models the choreography displayed in Figure 2.3a.

constructs.

WS-CDL has been attracting an imposing amount of attention in the research community clustered around choreographies. Among others, formalizations of the operational semantics of WS-CDL have been provided based on timed automata [91], several flavors of process calculi [207, 208, 212, 204, 130, 66, 65], Petri nets [194, 177], temporal logics [197], Control Flow Graphs (CFGs) [213] and Event-B [126] (a formal method for system-level modeling and analysis [4]).

However, despite all the interest garnered in the research community, WS-CDL has never really gained a foothold in the practice, which is demonstrated (and very likely also caused) by the almost complete lack of support and integration by the major SOA vendors of tooling and middleware. Indeed, the second implementation came only two years after the publication of the Candidate Recommendation [131], which is also touted to be the reason why WS-CDL has remained until now a W3C Candidate Recommendation: “W3C publishes a Candidate Recommendation to gather implementation experience” [196], which, in the case of WS-CDL, was never sufficiently gathered.

To the best of this author’s knowledge, the only comprehensive tooling for WS-CDL to date is SAVARA¹², developed in the scope of the JBoss Community. The fact that even SAVARA seems to be abandoning WS-CDL in favour of BPMN v2.0 Choreography Diagram (the support of which is the main goal of the SAVARA 2.x codebase) is very likely the writing on the wall for WS-CDL. The other rather famous project focusing on tooling for WS-CDL is pi4soa¹³, which has last released version 2.1.0 General Availability in August 2009; since then there seems to be no further ongoing

¹²SAVARA website: <http://www.jboss.org/savara>

¹³pi4soa website: <http://sourceforge.net/projects/pi4soa/>

```

1 <choreography name="ConsumerRetailerChoreography">
2
3   <relationship type="tns:ConsumerRetailerRelationship"/>
4
5   <variableDefinitions>
6     <variable name="purchaseOrder"
7       informationType="tns:purchaseOrderType"
8       silent="false" />
9     <variable name="purchaseOrderAck"
10      informationType="tns:purchaseOrderAckType" />
11     <variable name="retailer-channel"
12      channelType="tns:RetailerChannel"/>
13     <variable name="consumer-channel"
14      channelType="tns:ConsumerChannel"/>
15     <variable name="badPurchaseOrderAck"
16      informationType="tns:badPOAckType" />
17   </variableDefinitions>
18
19   <interaction name="createPO"
20     channelVariable="tns:retailer-channel"
21     operation="handlePurchaseOrder" >
22     <participate relationshipType="tns:ConsumerRetailerRelationship"
23       fromRoleTypeRef="tns:Consumer" toRoleTypeRef="tns:Retailer"/>
24     <exchange name="request"
25       informationType="tns:purchaseOrderType" action="request">
26       <send variable="cdl:getVariable('tns:purchaseOrder',' ','')"/>
27       <receive variable="cdl:getVariable('tns:purchaseOrder',' ','')"
28         recordReference="record-the-channel-info" />
29     </exchange>
30     <exchange name="response"
31       informationType="purchaseOrderAckType" action="respond">
32       <send variable="cdl:getVariable('tns:purchaseOrderAck',' ','')"/>
33       <receive variable="cdl:getVariable('tns:purchaseOrderAck',' ','')" />
34     </exchange>
35     <exchange name="badPurchaseOrderAckException"
36       faultName="badPurchaseOrderAckException"
37       informationType="badPOAckType" action="respond">
38       <send variable="cdl:getVariable('tns:badPurchaseOrderAck',' ','')"
39         causeException="tns:badPOAck" />
40       <receive variable="cdl:getVariable('tns:badPurchaseOrderAck',' ','')"
41         causeException="tns:badPOAck" />
42     </exchange>
43   </interaction>
44 </choreography>

```

Figure 2.10: A sample WS-CDL choreography [195].

development.

Several reasons have been pointed out for the lack of success of WS-CDL. First of all, the WS-CDL specification fails to provide an explicit meta-model and formal operational semantics for the constructs of the language [28, 30]. It has been claimed that π -calculus constitutes the formal foundation of WS-CDL, but this is a myth conclusively debunked: the constructs of the two languages do not directly match [45] and, besides, there is no normative mapping in the specification from WS-CDL to π -calculus constructs.

Another serious limitation of WS-CDL is that message-based interactions are exclusively on a one-to-one basis, while it has been argued that multi-participant interactions are fundamental both in the scope of service choreographies (see e.g. [30, 79]) and, more generally, on service-based interactions [29]. The lack of a graphical notation for WS-CDL and its rather unwieldy eXtensible Markup Language (XML) syntax has also attracted critiques [28].

Nevertheless, the biggest limit to the adoption of WS-CDL in the practice may be that its constructs badly fit WS-CDL's ideal orchestration counterpart in SOA, namely WS-BPEL. Many attempts to reconcile the two service composition languages have been made over the years (see, e.g., [91, 145, 208, 117, 146]); however, no work seems to completely and conclusively achieve the goal.

2.2.2.4 BPMN 2.0 Choreography Diagrams

With respect to its predecessor Business Process Modeling Notation (BPMN v1.x) [164], which focuses on orchestrations and interconnection choreographies, BPMN v2.0 [163] introduces¹⁴ the possibility of modeling interaction choreographies using the so-called Choreography Diagrams. An example of BPMN v2.0 Choreography Diagram is shown in Figure 2.11 (which has also been used in Chapter 1 as running example). BPMN v2.0 has both an XML based serialization format, which allows to model all the constructs offered by the language, and a graphical notation, which allows to represent a subset of the information that can be specified in the XML based format.

Like the rest of BPMN v2.0, Choreography Diagrams boast a very large number of constructs available to the modeler and, according to analyses based on understandability and completeness criteria [104, 76, 75], BPMN v2.0 may even be *too* complicated and construct-rich. Figure 2.12 and Figure 2.13 display the BPMN v2.0 constructs that are used in Choreography Diagrams to specify the messaging behavior of the choreography; other BPMN v2.0 constructs that can be used in Choreography Diagrams, like **groups** or **annotations**, do not have operational semantics and are here omitted. In a nutshell, BPMN v2.0 Choreography Diagrams are workflows made of **choreography tasks** (i.e., activities that represent message exchanges), **sequence flows**, **gateways** that specify how multiple sequence flows are combined and **events** like the beginning, completion and termination of choreography enactments. BPMN v2.0 choreography tasks allow to specify both one-to-one (Figure 2.12i) and one-to-many (Figure 2.12j) message exchanges (i.e., there can be multiple recipients for one message). Additionally, a choreography task can have a marker that specifies whether that choreography task is to be repeated multiple times sequentially, both as the same instance (Figure 2.12l) or as separate message exchanges (Figure 2.12n), or in parallel (Figure 2.12m).

With respect to this thesis’s terminology, **participants** in BPMN v2.0 actually correspond to roles. The BPMN v2.0 equivalent of “participant” as adopted in this thesis is **ParticipantEntity**. Participant entities are not graphically represented, but are instead only part of the XML-based serialization. In BPMN v2.0 Choreography Diagrams, participants in choreography tasks can either identify precisely one participant entity (the default case), or one participant entity out of a group, i.e., a **multi-instance participant** like the **Recipient** in Figure 2.12k. Which participant entity will actually take part in the enactment of the choreography is determined during the enactment, by the participants that will interact with them. For example, in the case of the choreography task

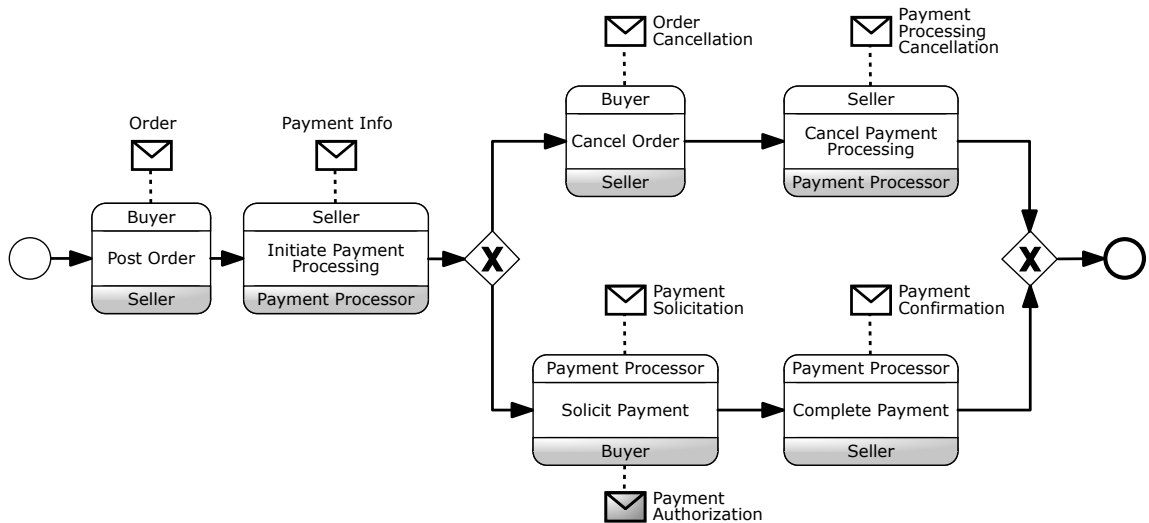


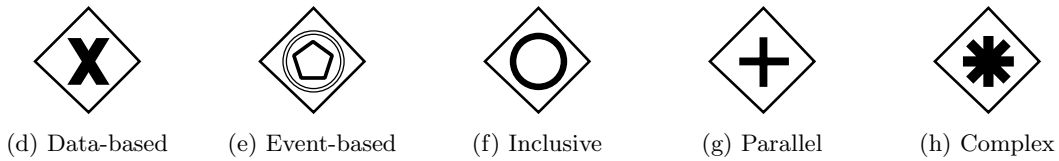
Figure 2.11: An example of BPMN v2.0 Choreography Diagram.

¹⁴Besides, of course, the rather baffling change in what the acronym “BPMN” stands for: from “Business Process Modeling Notation” in BPMN v1.x to “Business Process Model and Notation” in BPMN v2.0.

Sequence Flows



Gateways



Choreography Tasks & Nested Choreographies

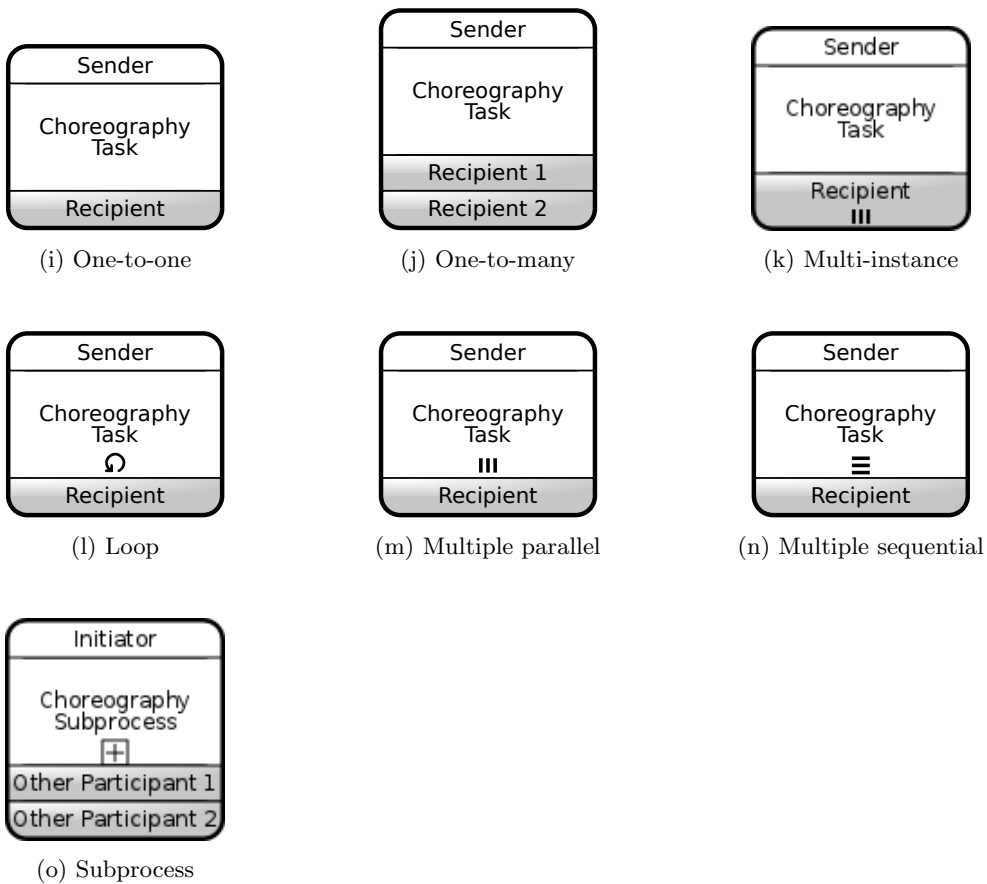


Figure 2.12: BPMN v2.0 Choreography Diagram constructs (Part 1).

represented in Figure 2.12k, the participant **Sender** will select which of the eligible participant entities will play the role of **Recipient**, and send the message to the respective endpoint.

Composition choreographies is supported in BPMN v2.0 via *nesting* and *referencing*. The nesting of a choreography inside another is supported by the **choreography subprocess** construct. Fundamentally, a choreography subprocess is a wrapper for a “sub-choreography” specified using the other BPMN v2.0 constructs. Choreography referencing is realized by **call activities**, which are fundamentally pointers to other choreography tasks or choreographies specified in different Choreography Diagrams (in the graphical notation, call activities are virtually indistinguishable from “normal” choreography tasks, save for a slightly thicker black border; they are not shown in Figure 2.12).

The ordering of choreography tasks and events is regulated by sequence flows and gateways. There are three types of sequence flows: the “normal” one (Figure 2.12a), which is activated as soon as its source element (e.g., a choreography task) has been completed, and the **conditional** and **default** sequence flows (Figure 2.12b and Figure 2.12c, respectively), which fundamentally have the same collective operational semantics of decision-based gateway, except that the condition is specified on the various conditional sequence flows, instead that on the gateway itself.

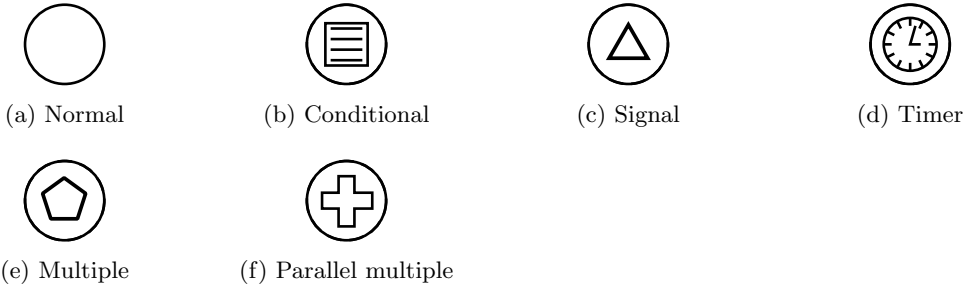
There are five types of gateways that can be used in Choreography Diagrams: data-based (Figure 2.12d) and event-based (Figure 2.12e) gateways represent decisions that result in the activation of one of their outgoing sequence flows. **Inclusive** gateways are used with conditional and default sequence flows as an alternative to data-based gateways. **Parallel** gateways activate all their outgoing sequence flows, hence allowing to start multiple “threads” in the choreography. Finally, **complex** gateways can fundamentally have any type of logic, which is specified by the modeler depending on the needs.

Figure 2.13 show-cases the very large variety of events that can be specified in Choreography Diagrams. Events are grouped in four categories. **Start events** are used to represent the beginning of an enactment. The different types of events are triggered in different situations. The “normal” start event (Figure 2.13a) is a placeholder that does not have a distinct operational semantics; instead, it is considered triggered when the constructs connected to it via sequence flows are enacted. The order types of start events have specific triggering conditions like data and time-based conditions (Figure 2.13b and Figure 2.13d, respectively), the section of a predetermined signal (Figure 2.13c), or combinations of other events (Figure 2.13e and Figure 2.13f). **End events**, instead, represent the completion of an enactment. The events that can appear in middle of the control flow of Choreography Diagrams, called **intermediate events**, are further split into two categories: **throwing** and **catching**. In a nutshell, throwing events are “active,” meaning that when a throwing event is reached, an event of the corresponding type is thrown. Thrown events are caught by catching events on basis of the type of the event and, in some cases, additional conditions. The disparity between types of throwing and catching events (three of the first, eleven of the second), is because most throwing events can be thrown only “within participants,” i.e., they can only be specified in BPMN v2.0 Process and Collaboration Diagrams (which are fundamentally orchestrations and interconnection choreographies, respectively).

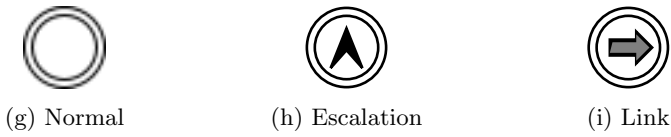
Like other industrial languages, BPMN v2.0 lacks a formal specification of the operational semantics of its constructs, offering only a natural language specification. This is particularly unfortunate in the case of BPMN v2.0 because of its extremely high complexity: the language offers a very large variety of constructs that interact with each other in very complex (and, one may think, rather underspecified) manners. Like with other industrial modeling languages, the research community is at work to provide formalizations of the operational semantics, e.g., by means of graph-rewriting rules [92]; as in the other cases, however, these efforts have no normative value and, therefore, do not necessarily improve the interoperability between BPMN v2.0 implementations. Presumably due to its complexity, the part of the BPMN v2.0 specification that deals with Choreography Diagrams is affected by a number of extremely serious defects¹⁵. For example, despite the fact that the specification shows a number of examples displaying intermediate catch events

¹⁵This author has personally submitted some of those issues, a few of which in collaboration with Oliver Kopp, from the Institute of Architecture of Application Systems (IAAS), University of Stuttgart

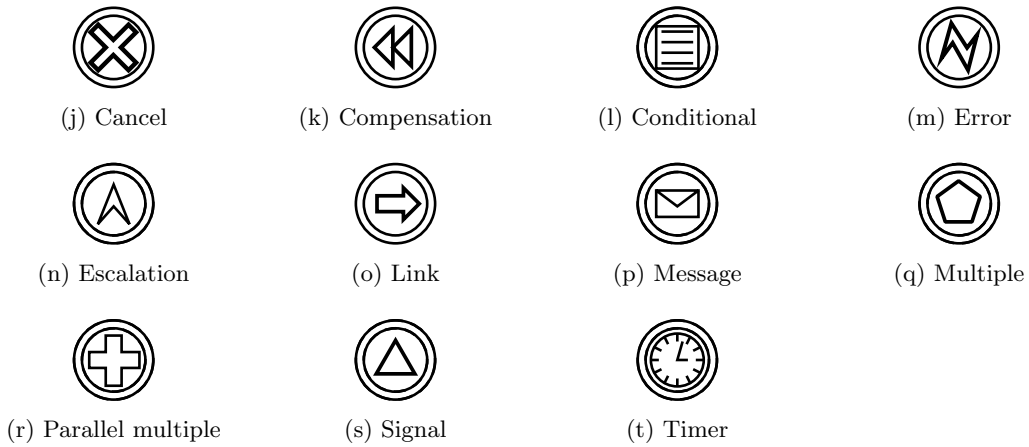
Start Events



Intermediate Throwing Events



Intermediate Catching Events



End Events



Figure 2.13: BPMN v2.0 Choreography Diagram constructs (Part 2).

attached to choreography tasks (instead of being connected by means sequence flows), the XML serialization does not support the actual attachment of events to choreography tasks. Moreover, issues with modeling interactions with Web Services in BPMN v2.0 have also been identified [108]. As per April 2014, there are 363 issues listed in the BPMN v2.0 “bug list” of the OMG¹⁶, of which approximately 34 applied to the choreography-related part of the specification. Unfortunately, as per June 5, 2013, the task force charted with the fixing of these issues is apparently dormant¹⁷, and it is unclear if, when and how the issues affecting BPMN v2.0 will be tackled.

2.3 Realizability of Choreographies

The goal of this section is to provide an overview of the state of the art of definitions of choreography realizability and the associated realizability analysis methods. Section 2.3.1 provides a taxonomy of styles and dimensions of realizability definitions. Section 2.3.2 defines a taxonomy of realizability analysis methods, which is applied to the state of the art in Section 2.3.3 and Section 2.3.4.

2.3.1 Categorization of Choreography Realizability Definitions

The problem of deciding the realizability of distributed messaging behaviors (also referred to as enactability [88], enforceability [211], local implementability [129] and well-assertedness [41]) has been studied since the 80’s (see, e.g., [3, 167]) for a large variety of modeling languages and formalisms.

The diversity of approaches to realizability analysis has lead over time to the accumulation of a large amount of often subtly, mutually inconsistent definitions. For instance, consider the various notion that have been named *weak realizability* by different authors. In the body of work about realizability of MSCs, the definition of weak realizability requires the composition of peers to be language equivalent to the choreography (i.e., the composition can enact all and only the conversations specified by the choreography), but not necessarily deadlock-free (see, e.g., [13, 157]); a similar notion of weak realizability is proposed in [52] for choreographies modeled as UML 2.x collaboration diagrams. However, weak realizability as defined in [116] is something very different. It is a form of realizability that requires language equivalence between the choreography and the composition of the peers under a relaxed notion of conversation equivalence: as long as two message exchanges involve different participants, they relative order is irrelevant when comparing conversations. And again, in [191] a choreography is said to be weakly realizable if the composition of the peers is able to enact only a certain subset of all the conversations that are specified by the choreography. Obviously, these three definitions of weak realizability are irreconcilable with one another. To further add to the confusion, weak realizability as defined in [191] is actually equivalent to *partial realizability* as defined in [135] and *local enforceability* in [211, 81].

The reason for such a fragmentation in the state of the art is likely that there are actually uncounted possible definitions of realizability depending on the assumed choreography modeling language or formalism (and, above all, its expressiveness), the goals to be achieved through realizable choreographies (projecting peers, ensuring correctness, etc.), the assumed communication models as well as, very interestingly, the associated realizability analysis method. Since it is obviously unfeasible to provide a consistent hierarchy of realizability definitions, in the remainder we endeavor to understand the technical reasons¹⁸ that have lead to so many different definitions. To this end, we are proposing two related taxonomies for definitions of realizability: *styles* (Section 2.3.1.1) and *dimensions* (Section 2.3.1.2).

¹⁶<http://www.omg.org/issues/bpmn2-rtf.open.html>, last accessed on 27 April 2014.

¹⁷See: <http://brsilver.com/is-bpmn-3-0-on-its-way>

¹⁸As opposed to the “human” ones, like the unavoidably incomplete knowledge of the state of the art. There are so incredibly many works about or connected with choreography realizability that this author has been very queasy for a very long time about writing up this particular “related work” section.

2.3.1.1 Styles of Realizability Definitions

One of the aspects that complicates the comparison of different definitions of realizability in the state of the art is that there are different styles of formulating the conditions under which a choreography is realizable; these style, the hierarchy of which is shown in Figure 2.14, apply both to formal and natural language definitions.

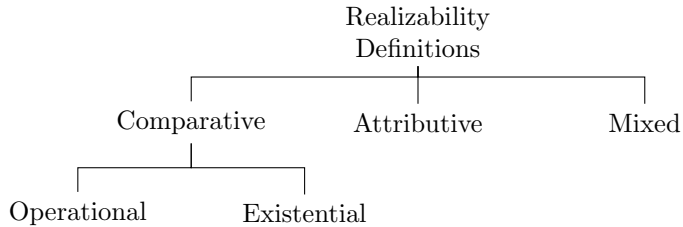


Figure 2.14: Hierarchy of the styles of definitions of realizability.

Comparative definitions are centered, like the name suggests, on the comparison between the messaging behaviors of the choreography and of the composition of the peers. This comparison is based on some notion of behavioral similarity, which is not necessarily an equivalence relation (i.e., a reflexive, symmetric and transitive relation). For example, the already-mentioned definition of weak realizability proposed in [191] requires only a subset of the conversations to be enactable by the composition of the peers. Set inclusion is antisymmetric and, therefore, not an equivalence relation. Nevertheless, equivalence relations seem to be by far predominant in the state of the art, and less constraining similarity relations tend to be perceived as “weak” or “relaxed.”

Comparative definitions can be further divided in *operational* and *existential*.

Operational definitions require that *it must be possible to effectively project the peers from the choreography*. An example of operational definition is the following one, enunciated in [116]:

“Particularly relevant is the problem of realizability of the choreography specifications, that is, the possibility to automatically extract from the choreography the behavioral skeletons of the participants [i.e., the peers] so that the concrete implementations [i.e., the participant implementations], built on the basis of these skeletons, are guaranteed to satisfy the choreography specification.”

Operational definitions are solidly grounded in the practical use of choreographies and their lifecycle (see Section 2.1.2): a realizable choreography is not “just” a specification of a distributed messaging behavior: it is a specification can be used to streamline the process of implementing said system by projecting skeletons for the participant implementations.

Existential definitions are rather more abstracted from the practical details of utilizing a choreography in the scope of a software development process. In fact, they limit the requirements to the theoretical existence of peers, the composition of which is behaviorally similar to the choreography. For example, consider the following definition of realizability presented in [13]:

“An MSC-graph is realizable if there is a distributed implementation that generates precisely the behaviors in the graph.”

Notably, this last definition contains neither word nor implication about *how* should the distributed implementation be attained. In a sense, existential definitions are akin to the \exists operator (“exists”) of predicate logic: for a choreography to be realizable, it must exist peers that, once composed, have a collective messaging behavior that is similar to that of the choreography; it is not required that said peers can be projected from the choreography itself. In this author’s opinion, existential

definitions are more elegant than operational ones, as they do not tie an intrinsic characteristic of choreographies such as realizability with practical aspects like the existence and nature of means of projecting the peers. Nevertheless, there is a lot to be said for the concreteness and pragmatism that shines through operational definitions.

Attributive definitions of realizability are radically different from comparative ones. Instead of relying on a comparison between the choreography and a composition of peers, they are formulated on the basis of properties that pertain exclusively to the choreography.¹⁹ For example, consider the following definition enunciated in [100]:

“A Büchi conversation protocol is realizable if it satisfies the lossless join, synchronous compatible, and autonomous conditions.”

The “lossless join, synchronous compatible and autonomous” conditions are defined in [100] on the basis of the assumed choreography modeling formalisms as logical predicates to be verified on the structure of choreographies. Attributive definitions are of particular interest, as they tend to *break down* realizability into separate, albeit correlated, properties of the analyzed choreographies. These properties can usually be verified separately from each other, which underlines the fact that verifying the realizability of a choreography is a multi-faceted, complex endeavor.

In a sense, every attributive definition may be seen as an *implicit existential definition*. Through the properties required of realizable choreographies, the attribute definition implicitly specifies criteria for behavioral similarity between the messaging behaviors of the choreography and of the composition of hypothetical peers.

Mixed definitions of choreography realizability that are partly comparative and partly attributive. Consider, for example, the following one provided in [175] that, paraphrased from its formal definition into natural language, runs as follows:

“A choreography is realizable if the composition of the projected peers is language equivalent to the choreography and it never deadlocks.”

In its original formulation, the attribute of deadlock-freeness is required of the composition of the peers. However, since the choreography and the composition of the peers are language equivalent, the choreography is necessarily deadlock-free as well. In this case, setting the requirement of deadlock-freeness on the composition of the peers instead than on the choreography is likely due to practical concerns: the verification framework used in [175] can perform out-of-the-box deadlock-freeness verification on the peers, while the latter would require additional modeling to be performed on the choreography instead.

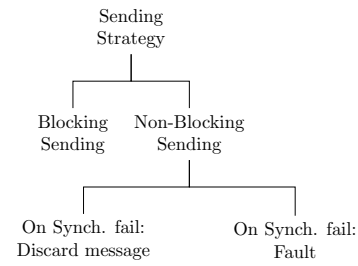
2.3.1.2 Dimensions of Realizability Definitions

Besides the categorization in styles presented in Section 2.3.1.1, definitions of choreography realizability are also characterized by the following *dimensions*:

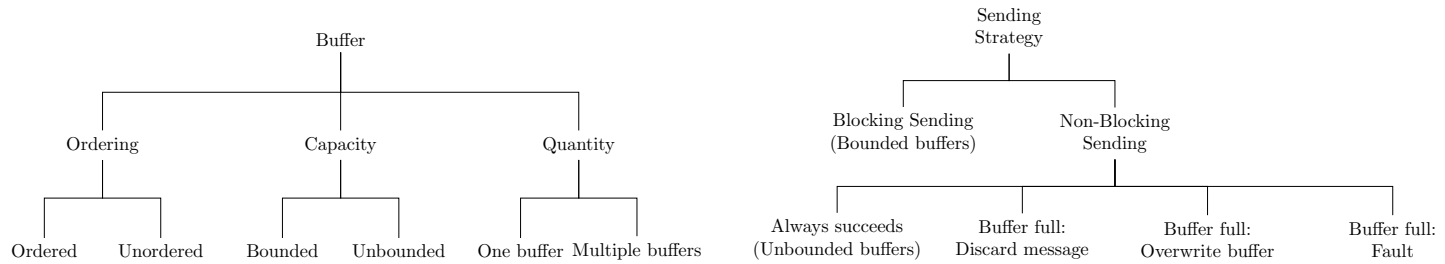
Choreography Modeling Language: Which choreography modeling language or formalism is assumed for the specification of choreographies.

Communication Model: The characteristics of the messaging communication that is assumed among the roles, e.g., synchronous messaging or asynchronous messaging with or without finite/infinite queues, the extent of reliability of the message exchanges (are all messages always received? are the messages received by the recipients in the same order the sender dispatch them?), the *arity* of the message exchanges (one-to-one, one-to-many), etc. In

¹⁹The name of this category is based on the meaning of “attributive” as “relating to an attribute” [73]: specifically, we refer to attributes of realizable choreographies (besides, of course, being realizable).



(a) Synchronous communication models.



(b) Asynchronous communication models.

Figure 2.15: Hierarchy of dimensions for communication models proposed in [132].

the remainder, the communication models are classified on the basis of the the taxonomy proposed in [132], which is summarized in Figure 2.15.²⁰

Peer Projection Method: How the peers are projected from the choreography.

Peer Composition Method: How the peers are composed with each other.

Behavioral Similarity Relation: Which type of relation is used to compare the behavior of the choreography with that of the composition of the peers, e.g., language or trace equivalence on the set of conversations, or bisimulation.

Intrinsic Properties: Which other properties of the choreography are required in order for it to be realizable, e.g., well-formedness, deadlock-freeness or stuck-freeness.

This thesis is not the first work to investigate the dimensions of definitions of choreography realizability. In [143], the author mentions the communication model (albeit apparently focused exclusively on which types of messaging queues are adopted), “conditions on the form of the choreography itself,” i.e., our intrinsic properties, and “the mechanical process used to extract the participant behavior specifications [i.e., the peers] from it,” which corresponds to our peer projection method.

Another, more comprehensive catalog of dimensions of realizability definitions is proposed in [79] and it is articulated over “communication model,” “complete behavior vs. subset of behavior” and “equivalence notion.” The “communication model” dimension of [79] and our namesake are fundamentally equivalent, albeit in this thesis are explicitly specified more fine-grained means of classifying the different communication models by means of the taxonomy provided in [132]. Our “behavioral similarity relation” dimension subsumes the “equivalence notion” and “complete behavior vs. subset of behavior” proposed in [79]. First of all, we have already shown that not all behavioral similarity relations at the basis of realizability definitions are equivalences, which alone warrants a reconsideration of the “equivalence notion” dimension. Moreover, the “equivalence notion” and “complete behavior vs. subset of behavior” dimensions seem to be tailored too closely on behavioral similarity relations that are based on set-theoretic comparisons the conversations, e.g., language and trace equivalence. This “set-centric” point of view does not accommodate equally well other behavioral similarity relations that build on top of relations among states, such as the various types of bisimulation. Finally, the “complete behavior vs. subset of behavior” proposed in [79] does not account for cases in which the composition of the peers is allowed or even expected to be able to enact conversations that are not specifically mandated by the choreography. This, for example, is necessary to define realizability in choreography modeling languages that, like BPMN v2.0 Choreography Diagrams, allow to model *open choreographies*, i.e., choreographies that specify only a subset of the conversations that the participants may be allowed to enact. That is, open choreographies allow participants to perform, in the scope of the enactments, additional message exchanges that are not explicitly mandated or even modeled, which means that the actual messaging behavior of the participant implementation can be a superset of that specified by the choreography.

As shown in Figure 2.16, the dimensions of definitions of choreography realizability are strongly correlated with the styles of realizability definitions that are identified in Section 2.3.1.1. In particular, the four styles reflect different subsets of the taxonomy of dimensions. (The comparative style is not considered, instead preferring to it its two subtypes operational and existential.) Notice that there is a “minimum common denominator” shared by all styles: choreography modeling language and communication model. This is to be expected: the definitions of choreography realizability are inextricably intertwined with the assumed choreography modeling language and the communication model, as they collectively define the complexity of the possible sets of conversations that can be modeled (or, in other words, the “formal expressiveness” of the choreography modeling language).

²⁰An alternative classification of communication models for services compositions has been proposed in [118], but its granularity is coarser than the one provided in [132].

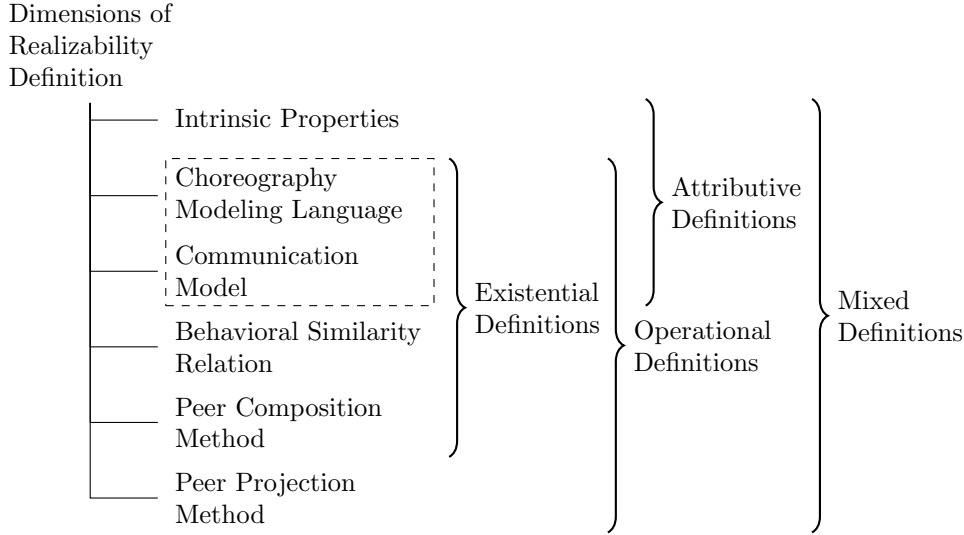


Figure 2.16: The dimensions of realizability definitions and their relation with the types of realizability definitions. The order of the dimensions has been rearranged to better fit the grouping in relation with styles. The dimensions surrounded by the dashed rectangle are shared by all the styles.

In addition to the dimensions shared by all styles, existential definitions address the behavioral similarity relation and peer composition method. Since existential definitions do not make assumptions on how the peers are devised, they do not cover the peer projection method. The peer projection method, instead, is covered by operational definitions in addition to all the dimensions already covered by existential definitions.

Attributive definitions add only the intrinsic properties on top of the common base. This is natural, considering that the behavioral similarity relation and the peer projection and composition methods are directly related with the peers, which are not involved in attributive definitions.

Finally, mixed definitions span across all dimensions. The only possible caveats are the peer projection and peer composition methods, which are addressed only when the mixed definition adopts an operational approach rather than an existential one.

2.3.2 Categorization of Realizability Analysis Methods

Realizability analysis methods can be divided in the following categories:

Constructive methods are called so because they involve “constructing” the peers by means of a projection process. Constructive methods all follow the same canvas:

1. Project the peers from the choreography;
2. Compose the peers;
3. Compare the behavior of the composed peers with that of the choreography.

Depending on the outcome of the comparison, the choreography is judged to be realizable or unrealizable.

Constraintive methods consist of verifying intrinsic properties of the choreography by means of constraints that, if satisfied, prove that choreography’s realizability. In some cases, such as [100], the verification of the intrinsic properties is a sufficient but not necessary condition. That is, such realizability analysis methods admit *false positives*, i.e., choreographies that, while actually realizable, are judged to be unrealizable by the analysis method.

Mixed methods are both constructive and constraintive, because after the comparison between the behavior of the composed peers and that of the choreography, intrinsic properties are also verified.

The parallel between the taxonomy of styles of definitions of choreography realizability discussed in Section 2.3.1.1 and the one above is straightforward: constructive realizability analysis methods correspond to comparative definitions (and, more specifically, to operational definitions), constraintive ones correspond to attributive definitions and mixed methods correspond to mixed definitions. It should be noted, however, that the correspondence between styles of definitions and types of analysis methods is not a rule and, in fact, is not always followed in the state of the art. For instance, the authors of [52, 51], discussed later in Section 2.3.4.1, combine a mixed definition with a constraintive analysis method: by means of mathematical proofs it is then shown that if the constraints they specify are met, peers that realize the choreography *can be projected*.

There is, however, one combination of styles of realizability definitions and types of analysis methods that, in this author’s opinion, is unlikely ever to be found in the state of the art: attributive definitions coupled with a constructive analysis methods. This is due to the following technical reason: projection is generally not a one-to-one function. In principle, one may project many syntactically different peers for a given role. Therefore, it seems far harder (and, depending on the choreography modeling languages, possibly unfeasible) to prove intrinsic properties of a choreography by means of its projected peers (i.e., attributive definition verified by means of constructive analysis method) than to prove the existence of some adequate peers given intrinsic properties of the choreography (i.e., comparative definition verified through a constraintive analysis method).

In the remainder, the state of the art of constructive and mixed analysis methods is covered in Section 2.3.3; the constraintive analysis methods in are treated in Section 2.3.4.

2.3.3 Survey of Constructive and Mixed Realizability Analysis Methods

Constructive approaches to realizability analysis are all rather similar to each other. The steps are clear: project the peers, compose them and compare the behaviors of the composed peers and the one of the choreography. Actually, most of the specificity of any constructive analysis method is captured by the dimensions of the choreography realizability definition it aims at verifying. In the case of the mixed ones, there is the “extra twist” of the intrinsic properties that are verified. However, they seem to invariably be forms of stuck-freeness, deadlock-freeness or progress. Table 2.1 presents a categorization of the constructive and mixed analysis methods found in the state of the art in terms of the dimensions that are covered by constructive and mixed choreography realizability definitions (see Section 2.3.1.2), namely: choreography modeling language, communication model, peer projection, peer composition, behavioral similarity relation and intrinsic properties.

By surveying the state of the art, we found a distinct lack of details about the assumed communication model (an issue also noted by the authors of [132]). Whenever a clear match could be found in the description of the communication model, the most likely missing details were inferred; in such cases, the “???” marking is applied to the corresponding entry.

At first look, the various realizability analysis methods are very diverse in terms of choreography modeling languages, communication models and behavioral similarity relations. The composition of the projected peers is also very diverse and strongly depends on the type of choreography modeling language and model-checking tools that are utilized. Each approach to the composition of peers has all the nuances and assumptions that one would expect from a topic under research scrutiny for more than fifteen years; for reasons of space, we cannot cover these particulars in detail.

There is markedly little diversity, however, insofar peer projection methods are concerned. The predominant approach to projection of peers is the so-called “natural projection,” i.e., the transformation of the choreography into a peer by changing the message interactions and decisions as to reflect one role’s perspective, and removing all actions that do not involve that one role. Each and every one of the approaches classified in Table 2.1 uses some form of natural projection. For example, in the case of [81], the outcome of the natural projection is refined by removing empty

Table 2.1: Classification of constructive realizability analysis methods.

Source	Choreography Modeling Language	Communication Model	Peer Projection	Peer Composition	Behavioral Similarity Relation	Intrinsic Properties
Decker and Weske [81]	Interaction Petri nets	Synchronous (???)	Natural projection plus reduction rules to remove empty places	Wiring of peer nets via additional transitions connecting “sending” and “receiving” places	Branching Bisimulation [105]	<i>none</i>
Roohi, Salaün, and Mirian-Hosseinabadi [175]	Process algebra without channel passing	Synchronous: blocking sending (???)	Natural projection	Synchronization of sending and receiving activities using channels	Language equivalence	Deadlock-freeness of composed peers
Hallé and Bultan [109]	Conversation protocols (based on STSs)	Asynchronous: non-blocking sending: always succeeds buffer: ordered (???) , unbounded, multiple buffers (one per participant)	Natural projection plus annotation of expected states of other participants	Cartesian product of the states of the peers	Language equivalence	<i>none</i>
Poizat and Salaün [168]	(subset of) BPMN v2.0 Choreography Diagrams	Synchronous (???) & Asynchronous: non-blocking sending: always succeeds buffer: ordered (???) , bounded, multiple buffers (one per participant)	Natural projection on FSMs representing the choreography constructs	Additional LOTOS process that synchronizes corresponding, projected construct states in the peer processes	Strong equivalence [151]	<i>none</i>

Continued on next page

Continued from previous page

Source	Choreography Modeling Language	Communication Model	Peer Projection	Peer Composition	Behavioral Similarity Relation	Intrinsic Properties
Salaün, Bultan, and Roohi [179]	UML 2.x Collaboration Diagrams	Synchronous: blocking sending & Asynchronous: non-blocking sending: always succeeds buffer: ordered, bounded, multiple buffers (one per participant)	Natural projection	Additional LOTOS process that synchronizes peers on shared messages	Strong equivalence [151]	<i>none</i>
Basu, Bultan, and Quederni [35]	Conversation protocols (based on FSMs)	Asynchronous: non-blocking sending: always succeeds buffer: ordered, unbounded, multiple buffers (one per participant)	Natural projection	Additional LOTOS process that synchronizes peers on shared messages	Language equivalence	<i>none</i>

constructs (those to which the projected role does not participate to). In the *shared-state projection* proposed in [109], the classic approach to natural projection is compounded by annotating the states of the projected peer with the expected, corresponding states of other participants.

Quite possibly, all surveyed constructive and mixed realizability analysis methods share ties to natural projection for reasons of *convenience*. Natural projection is a very straightforward way of defining projection operators. For those choreography modeling languages for which natural projection is easy to define, a constructive realizability analysis method is, well, a natural fit²¹. Therefore, it is probably not the case that natural projections are preferred over other projection methods for any particular reason: it is more likely that constructive methods emerge when natural projection is easy to define. After all, given the projection and composition operators, the only “missing ingredients” are the peer composition method (and we have more than a decade of research on service composition to build upon) the verification of a behavioral similarity relation (here, the decades of research are at least three). Of course, not all realizability analysis methods that focus on choreography modeling languages with easily-defined natural projection are constructive, but the correlation appears undeniable.

At last, it is necessary to mention what *is not* part of this classification and that, in this author opinion, definitely *should be*. When first drafting up this classification, an additional classification criterion had been envisioned: computational complexity. That is, an estimation of the computational complexity of the overall realizability analysis method. Computational complexity does not appear in this classification because, very surprisingly, none of the surveyed publications provided any information about it. What is provided in [175, 109, 168, 179, 35] are measurements of the performance of the methods given various combinations of amounts of roles and message exchanges among them. While such “benchmarks” are meant to validate the viability of the proposed realizability analysis methods, they do not enable any direct comparison among them. Different publications test performances on different choreographies, which gives no guarantee that different realizability methods “scale” comparably.

2.3.4 Survey of Constraintive Realizability Analysis Methods

Unlike constructive and mixed realizability analysis methods, which are all very similar in the general approach they follow, constraintive ones are hardly classifiable in a simple, coherent framework due to the strongly ties to the particular choreography modeling languages they assume. In the remainder of this section, constraintive realizability analysis methods are discussed grouped by similarity of approaches and underpinning concepts.

2.3.4.1 Awareness

Constraintive realizability analysis methods build on top of the concept of *awareness*. Awareness is a representation of the “knowledge,” almost always incomplete, that a participant has during the enactment about the current state of the enactment itself. Awareness is often modeled based on an event-centric view of choreographies. Depending on the constructs offered by the assumed choreography modeling language, events may represent the dispatching and reception of messages, the taking of decisions, the invitation of other participants via channel passing, etc. Depending on assumptions about the visibility of the actions performed by the participants, e.g., only participants that exchange messages know that the message exchange has occurred, participants are limited in which events they are *aware* of, i.e., which events the participants can observe. The fact that awareness of participants is generally limited to a subset of the events occurring in an enactment is known as *blindness* [88] or *myopia* [188]. Constraintive analysis methods based on awareness aim at verifying that the blindness of the participants does not prevent them from enacting the choreography correctly (i.e., without violations) and in its entirety (e.g., all conversations specified by the choreography can be correctly enacted).

²¹(Bad) pun totally intended.

Awareness in UML 2.x Collaboration Diagrams: In [52, 51], the authors propose a model of awareness that they call *well-informedness*. They model UML 2.x collaboration diagrams (see Section 2.2.2.2) as event-based systems in which events represent the dispatching of messages. Different types of events represent the dispatching of messages that can occur exactly once (*single message send event*), conditionally to some internal decision by a participant (*conditional message send event*) or one or more times subject to decision of the sender (*iterative message send event*). Using the taxonomy for communication models proposed in [132], message exchanges can be either synchronous (“blocking sending” strategy), or asynchronous (“always succeeds” strategy, one ordered, unbounded buffer per participant). The events are related with each other by a partial ordering relation (assumed to be acyclic in order to avoid deadlocks) among events, i.e., which events must occur before others can. Moreover, since collaboration diagrams are *threaded* (see Section 2.2.2.2), the events are partitioned according to the thread in which they occur.

The two following existential definitions of realizability are provided:

Weak realizability: A collaboration diagram is weakly realizable if exist peers, the composition of which is language equivalent to the collaboration diagram but can get *stuck*, i.e., there are some conversations that cannot be executed to the end because of issues of blindness;

Strong realizability: A collaboration diagram is strongly realizable if exist peers, the composition of which is language equivalent to the collaboration diagram and stuck-free.

The weak realizability of a collaboration diagram is verified using the following sufficient conditions (which are also shown to be necessary for collaboration diagrams in which send events are all single message send events):

Separation: Each event appears in exactly one thread; therefore, participants do not need to “guess” which thread an event belongs to;

Well-informedness: An event e is *well-informed* if either:

- The event e is an initial event;
- The event e is not initial and, for all its immediate predecessor events, either the predecessor event e' is a synchronous single message send event and the sender of e is either the sender of the recipient of e' , or e' is a conditional or iterative message send event and the sender of e and e' is the same participant.

A collaboration diagram with the separation property and in which all events are well-informed is shown to be strongly realizable (and thus also weakly realizable). In a nutshell, the constraints for the well-informedness of an event are simple rules that encode *how participants track the evolution of the state of the enactment through the observation of events*. The sender of a message exchange always knows that the message has been sent, as it can observe its own actions. If the message exchange is synchronous, the recipient also knows that the message has been sent the moment it is sent, as the recipient can observe its own action of reception. In case of asynchronous message exchange, there is no guarantee as to when will the recipient consume the event, and hence it cannot be relied upon that the recipient will keep track in a timely manner of how the enactment state changes.

Internal decisions are by definition observable only to the participant that performs them. This complicates matters considerably in the case of conditional and iterative message send events. Whether a message will be sent at all (in case of a conditional send event) or once more (in case of an iterative send event) is known to the recipient only when the message is received. A recipient will wait indefinitely for messages that do not get sent unless it receives at some point some other message that can occur only later in the sequence. Given that the message queues are ordered, that allows the recipient to imply that the sender of the first message must have decided against sending it, and therefore the wait for the recipient is over.

Awareness in STSs with Time Constraints: In [138], this author has proposed sufficient conditions for the realizability of STS with time constraints associated with the transitions based on awareness of both states and transitions. Considering that the reaching of the state and the firing of a transition can be seen as events, this approach is very similar to the one proposed in [52, 51]. However, the presence of time constraints on transitions requires an extra provision to guarantee the progress of choreography enactments: if a state has outgoing transitions constrained by means of time constraints that are not satisfied infinitely often (which implies that the choreography could get stuck in the state), then there must exist a silent transition that can eventually lead to another state.

2.3.4.2 Synchronizability

In [101, 103, 53], the authors adopt a comparative realizability definition for interaction choreographies specified as Büchi automata. The assumed communication model is asynchronous messaging with one unlimited, ordered queue for each peer. The realizability definition assumes natural projection as means of projecting the peers and language equivalence as behavioral relation. Instead of using a constructive realizability analysis method, these works adopt a constraintive approach based on verifying the following three sufficient (but not necessary) conditions:

Lossless join: The set of conversations specified by the choreography is equal to the one resulting from the cartesian product of the languages of the projected peers. It is very important to notice that the lossless join condition is not another formulation of the language equivalence of the choreography and the composition of the projected peers. In fact, the lossless join does not specify requirements on the final states reached by the choreography and the composition of the projected peers after the enactment of one conversation. Moreover, in the lossless join condition the conversation set of the choreography is not compared with the conversation set of the composition of the peers, but instead with the cartesian product of the languages of the single peers. The cartesian product of the languages of the single peers is in general a superset of the language of the composition of the peers (the cartesian product is the least “restrictive” way of combining the behaviors of the peers, as all possible combinations are allowed); therefore, the lossless join condition is a lesser behavioral similarity relation than trace equivalence (and therefore also of language equivalence).

Synchronous compatibility: A choreography is synchronous compatible if the composition of the projected peers is such that their cartesian product does not have “illegal states,” i.e., states that have no corresponding ones in the choreography. Illegal states result from lack of synchronization due to the asynchronous messaging. In other words, a choreography is synchronously compatible if the composition of the projected peers has the same behavior when assuming either synchronous or asynchronous messaging. Synchronizability has also been studied for choreographies specified using Petri nets [86] and FSMs [34, 36].

Autonomy: A choreography satisfies the autonomy condition if each of its projected peers, at any time during the enactment of a conversation, are in a state such that they can exclusively receive messages, send messages or terminate. In other words, the autonomy condition aims at ruling out situations in which a participant might violate the choreography by performing an action that is invalid due to a message already sent by another participant, but not yet received and consumed. For example, the choreography in Figure 1.1 does not have the autonomy property: the *Buyer* has a state in which it can either send the *Order Cancellation* or receive the *Payment Solicitation*; since these are mutually exclusive actions, the choreography suffers of a race condition that, due to asynchronous messaging, can lead to violations of the choreography.

2.3.4.3 Controllability

In [135], the authors present two definitions of realizability, *partial realizability* and *complete realizability* for choreographies specified as Open Workflow Nets. A choreography is partially realizable if there exist peers that, once composed, can enact a strict subset of the conversations specified by the choreography. For *complete realizability*, there must exist peers that, once composed, can enact all and only the conversations specified by the choreography. The authors reduce partial and complete realizability to a type of controllability for services with multiple partners called *decentralized controllability*. A peer is said to be controllable if it exists another peer so that their composition is deadlock-free [182]. The notion of decentralized controllability is built on top of the notion of *monitor*: a monitor for a choreography is a “virtual,” additional participant for that choreography defined as an Finite State Automata (FSA) that synchronously receives all messages that are sent by all other participants and reaches a final state if and only if the conversation enacted is one of those specified by the choreography. In other words, the monitor is a passive participant that simply checks if the enactment has violated the choreography or not. The choreography is said to be decentralized controllable if the composition of its monitor with automata that represent the participants is deadlock-free. These automata are not “proper” peers, in that they are not specified as to interact with each other; instead, they are specified in order to communicate exclusively with the monitor, which acts as a sort of “middleware” that routes the messages between automata.

2.3.4.4 Choice Soundness

In the literature about MSCs, trace equivalence is usually taken as the behavioral similarity relation for realizability definitions. Additionally, the composition of the peers must be deadlock-free, see e.g., [13]. The research community of MSCs has investigated in depth different types of types of realizability defects that fall under the collective name of *non-local choice* issues. In general, non-local choice is a modeling defect that may lead to violations during enactments because the participants evaluate independently the same condition, reach different results, and therefore perform mutually-incompatible parts of their behaviors [37]. MSCs, considered singularly, do not have choices; the compositions of multiple message sequence charts, called HMSCs, do have choices that establish which of the composed MSCs is to be executed next.

In [155, 156], the authors propose formalizations for non-local choice as well as for two other choice-related problems: *non-deterministic choice* and *race-choice*. Non-deterministic choice is the situation in which the partners of a participant performing a choice cannot understand which of multiple MSCs is being enacted because the latter are “too similar,” i.e. from the point of view of the participants, these MSCs cannot be differentiated. Race-choice, instead, is a realizability defect for which a message intended to denote the beginning of a new MSC is instead erroneously interpreted as part of another, concurrently enacted MSC. In this case, sender and recipient give a different interpretation of which behavior is being enacted, which results in discrepancies. If a HMSC has no issues of non-local choice, non-deterministic choice or race-choice, it is said to have *sound choice*.

2.4 Realizability-Driven Evolution of Choreographies

In the scope of this thesis, we define *choreography evolution* as follows:

Definition 2.2 (Choreography Evolution). Choreography evolution is the process of modifying at modeling-time existing choreographies through the sequential application of a series of changes with the aim of achieving specific modeling goals.

Modeling goals can be, for example, the correction of realizability defects, the improvement of QoS aspects of the enactments and the adjustment of the collective messaging behavior to emerging requirements (see, e.g., [137]). Evolution as intended in this work is often referred to as *static* in order to differentiate it from *dynamic evolution*, i.e., the modification at enactment-time

of the behavior to be performed by the participants [178] (also known as “reconfiguration” [180] or “instance migration” [199]).

One aspect should be kept in due consideration: choreographies do not generally evolve in isolation. If there are already participant implementations that have been realized to take part in the choreography, choreography changes may render them non-compliant. Therefore, pre-existing participant implementations that should enact the updated choreography must similarly evolve, and this has technical implications (changing software systems always involve risks) as well as economical and business-related ones (changes to software systems do not come for free: they required investing resources, which in the case of distributed systems tend to be considerable). In the state of the art there are approaches such as [202, 200, 111, 162] to (semi)automatically “propagate” changes applied to a choreography to the relative participant implementations, making them compliant once again.

In [14], changes to software services are characterized as either *deep* or *shallow*, depending on whether the changes affect the compatibility of the modified service with its partners. The same categorization can be applied to choreography changes, which are deep or shallow depending on whether or not they compromise the conformance of existing participant implementations. The present thesis is concerned with the evolution of choreographies in order to solve their realizability defects. Changes that solve realizability defects are necessarily deep: to solve a realizability defect, the messaging behavior of the choreography must be changed, which in turn compromises the compliance of some or all participant implementations.

The remainder of this section is organized as follows. The criteria that will be used to classify the approaches in the state of the art are presented in Section 2.4.1. The criteria are then applied to the existing approaches to the realizability-driven evolution of choreographies in Section 2.4.2.

2.4.1 Categorization for Realizability-Driven Evolution Methods

Approaches to correct realizability defects in choreographies consist of defining *remediation strategies*. A remediation strategy is an algorithm that, given a choreography with some realizability defect, calculates one or more *remediation changes*, i.e., modifications of the choreography that once apply to it, solve the realizability defects. In general, a change to a choreography can be of one of the four following types:

Additive changes consist of adding additional elements to the choreography. For example, “injecting” an additional choreography task and connected sequence flow between two pre-existing choreography tasks in a BPMN v2.0 Choreography Diagram is an additive change.

Subtractive changes are the opposite of additive changes: they consist of removing pre-existing elements from the choreography; no new elements are added.

Alterative changes do not add or remove elements, but instead they change the configuration of existing elements. For example, adding or removing a recipient from a choreography task of a BPMN v2.0 Choreography Diagram is an alterative change.

Mixed changes are complex changes that mix one or more simpler changes of two or more of the other types. For example, “swapping” the position of two choreography tasks in BPMN v2.0 Choreography Diagram is a mixed change, as it can be seen as first removing those elements (two subtractive changes) and then adding them in inverted positions (two additive changes).

Based on the above categorization of changes to choreographies, one can categorize the different types of remediation strategies according to the type of changes they generate. That is, additive strategies are those that generate additive remediation changes, subtractive strategies generate subtractive changes, and so on.

Of course, the remediation strategies are based themselves on the definition of realizability they aim at achieving. The following criteria will be used to categorize the approaches for evolving choreographies in order to solve realizability defects:

Realizability Definition: Which type of realizability is achieved by applying the proposed strategies. The realizability definitions will be characterized according to the styles (presented in Section 2.3.1.1) and the dimensions they cover (see Section 2.3.1.2).

Remediation Strategy Type: How does the strategy affect the choreography in terms of modifications to its elements.

Intervention Degree: Whether the remediation strategies are automatic or involve manual work by the modeler.

Similarly to the case of the classification of realizability analysis methods, we would very much have liked to add the computational complexity of the remediation strategies to the classification criteria. However, none of the works here surveyed treated this aspect, and thus we also omit it.

Another aspect of interest would have been how the remediation changes affect the scalability of the choreography. However, as noted in the systematic literature review of service choreography adaptation published as [127], the impact of the changes in terms of scalability of the choreography is another extremely neglected aspect in the state of the art.

2.4.2 Survey of Realizability-Driven Evolution Methods

Table 2.2 presents a categorization, based on the criteria outlined in Section 2.4.1, of the existing approaches to the realizability-driven evolution of choreographies. The approaches are sorted by chronological publication order. The approaches are rather diverse from each, partly due to the different choreography modeling languages they assume and partly due to the different strategies they employ. In the remainder of this section, these approaches listed in Table 2.2 are shortly surveyed.

Lekeas, Kloukinas, and Stathis [128]: This work assumes interaction choreographies modeled as FSMs and proposes a technique to correct realizability defects that are due to silent transitions, e.g., changes in status of the enactments that are not visible to some participants and that can lead, for example, to non-local choice issues (see Section 2.3.4.4). This approach is performed iteratively on all the roles: for each role, the respective peer is projected using natural projection 2.3.3). Natural projection likely produces a peer with *silent actions*, i.e., transitions from a state to another that are not visible to the participants playing the projected role. Adopting the terminology of the works based on the concept of awareness (see Section 2.3.4.1), the participants playing that role are not aware of the message exchanges that, through natural projection, become silent actions. In some cases, this lack of awareness does not cause realizability issues, e.g., when all the message exchanges following a silent action have the role as recipient: the participants playing the role will be able to “resynchronize” their knowledge of the state of the enactment upon receiving any of the messages. In less fortunate cases, however, unawareness may lead to problems like non-local choice. The authors of propose the following three alternative remediation strategies to correct such realizability defects:

Repair all silent actions: The participant is made aware of all message exchanges it could not originally see by adding it as recipient to them. This strategy is, so to say, the “nuclear option,” as it removes all silent actions for the participant, including those that did not cause realizability defects;

Repair “frontier” silent actions: This strategy is based on branching bisimulation: the states of the original choreography are grouped in equivalence classes with respect to the respective states in the peer. The role is added as message recipient to all those message exchanges that are invisible to the projected role and that lead to a state that belongs to a different equivalence class. That is, this strategy does not repair silent actions that lead the enactment to states that are projected to the same state in the peer. This strategy is a refinement of the first one, in that it tries not to modify the choreography in parts that do not affect the role currently considered.

Table 2.2: Classification of approaches for realizability-driven evolution of choreographies.

Source	Realizability Definition	Remediation Strategy Type	Intervention Degree
Lekeas, Kloukinas, and Stathis [128]	<p>Style: Attributive</p> <p>Choreography Modeling Language: FSMs</p> <p>Communication Model: Asynchronous (???)</p> <p>Intrinsic Properties: Absence of “structural problems” (???)</p>	Alterative	Automatic
Bocchi, Lange, and Tuosto [41]	<p>Style: Attributive</p> <p>Choreography Modeling Language: Global assertions (a session type calculus with additional constrains modeled as formulas)</p> <p>Communication Model: Asynchronous (???)</p> <p>Intrinsic Properties: History sensitivity & temporal satisfiability</p>	Alterative	Automatic
Salaün, Bultan, and Roohi [179]	<p>Style: Operational</p> <p>Choreography Modeling Language: UML 2.x Collaboration Diagrams</p> <p>Communication Model: Synchronous, blocking sending; Asynchronous, non-blocking sending (always succeeds), buffers ordered, unbounded, one per participant</p> <p>Peer Projection Method: Natural projection</p> <p>Peer Composition Method: Additional LOTOS process that synchronizes peers on shared messages</p> <p>Behavioral Similarity relation: Strong equivalence</p>	Additive	Automatic

Continued on next page

Continued from previous page

Source	Realizability Definition	Remediation Strategy Type	Intervention Degree
Lanese, Montesi, and Zavattaro [124]	Style: Existential Choreography Modeling Language: Process Algebra Projection type: Natural projection Behavioral Similarity relation: Trace equivalence	Mixed	Automatic

Repair only harmful silent actions: This strategy is a further refinement of the previous: given the branching bisimulation-based equivalence classes of the states of the choreography, the only repaired silent actions are those that:

- Lead the peer to states in the projected peer from which the peer can either receive or send messages;
- Lead the peer to states from which the peer can only send messages that are dispatched to different peers;
- Lead the peer to states from which the peer can only send messages and those message exchanges lead to different states in the peer.

The three remediation strategies are ordered according to an increasing level of “precision,” i.e., each is less likely than the previous to modify the choreography without actually solving a realizability issue.

In this author’s opinion, [128] suffers from a fatal flaw: the virtually complete omission of which definition of realizability is assumed and, thus, is assured by the proposed repair strategies. The only explicit explanation of what is meant to be accomplished is the following: “[to] ensure that there are no structural problems with the protocol that the agent receives, i.e., it is enactable.” In this author’s opinion, it is likely that the realizability definition implied by the authors is based on a behavioral similarity relation somewhere in between trace- and language equivalence (and, quite possibly, trace equivalence itself). This is because the proposed repair strategies aim at solving non-local choice issues, which compromise the trace equivalence (see, e.g., [37]); however, there are no provisions to ensure the the choreography reaches a final state if and only if all the peers do, which rules out language equivalence.

Bocchi, Lange, and Tuosto [41]: This work assumes choreographies modeled using session types (see Section 2.2.1.1). The content of the messages exchanges between the participants is modeled as variables containing numeric values, the values of which are constrained by assertions. The authors focus on two types of realizability defects:

Violations of history sensitivity, i.e., situations in which a participant has to send a message on basis of the unknown value of a variable previously defined by other participants (or, in other words, unawareness of the content of another message);

Violation of temporal satisfiability, i.e., when the value assigned to a variable by participant makes it impossible to satisfy some constraints later in the enactment.

To solve the realizability defects of history sensitivity, two alternative remediation strategies are proposed: *strengthening* and *variable propagation*. Strengthening consists of replacing of the variable unknown to the participant with one that is known. Variable propagation, instead, foresees the modification of some message exchanges preceding the one that violates history sensitivity so that the value of the unknown variable gets propagated to the participant that needs it.

To solve realizability defects of temporal satisfiability, the authors propose an alternative remediation strategy called *lifting*. Lifting consists in the modification of the assertions that regulate the possible values of the variable that causes the issue so that it is guaranteed that any valid value for this variable will not compromise the satisfiability of later interactions.

Salaün, Bultan, and Roohi [179]: This work tackles realizability issues in UML 2.x Collaboration Diagrams. As discussed in Section 2.2.2.2, UML 2.x Collaboration Diagrams are threaded. Messages in the same thread can be exchanged between different participants, and are ordered in sequences. As such, collaboration diagrams can have the following two types of realizability defects:

Violation of dependency relations between message exchanges in different threads;

Violations of sequence orders of messages within one thread.

To solve realizability defects of these two types, the authors propose two additive remediation strategies that based on the introduction of new types of message exchanges, called **SYNCH** and **SEQ**. **SYNCH** messages are introduced to enforce the dependency relations between message exchanges in different threads. **SEQ** messages are used to enforce the sequence order of messages within one thread. Two different communication models are considered: synchronous and asynchronous with bounded queues. Under the synchronous communication model, a **SEQ** message exchange must be added after one “normal” message exchange whenever the sender of the latter is neither sender nor recipient of the previous message exchange in the thread. In the asynchronous case, the recipient does not necessarily receive the message when the latter it is dispatched, and therefore a **SEQ** message between two message exchanges in one thread whenever the sender of one the first is not also the sender of the second. Analog conditions are specified for **SYNCH** messages, except that they refer to dependency relations among threads instead of message exchange order in one thread.

Lanese, Montesi, and Zavattaro [124]: The strategies proposed in this work are built on top of the idea of *connectedness* (which, for all practical purposes, seems to be another formulation of awareness), i.e., that the choreography is specified so that the participants can comply with causal and temporal dependencies between the choreography’s elements (e.g., ordering of message exchanges). Three related notions of connectedness are considered, concerning sequences, choices and repeated operations (i.e., iterations):

Connectedness of sequences foresees that in a sequential composition of two message exchanges, the second takes place only after the first has. The strategy proposed to solve issues with connectedness of sequences is alterative, and consists of changing the participants that sends the second messages to be the same that has received the first (and therefore has enough information to know when it is allowed to send the second message).

Connectedness of choices requires that “in a choice all the participants agree on the chosen branch” and it can be guaranteed by ensuring that “a single participant performs the choice and informs all the other involved participants.” The strategy proposed to solve issues with the connectedness of choices is alterative, and consists of changing the senders of all message exchanges that occur immediately after a choice to be the same participant.

Connectedness of repeated operations is violated when the choreography is so that one message pertaining to one message exchange is not by mistake associated with a different message exchange running in parallel (which is equivalent to the “race choice” discussed in [155]). The strategy proposed to solve this type of issues is to reduce the amount of parallelization in the choreography, introducing strict temporal dependencies between message exchanges that involve the same type of message. In a nutshell, this strategy consists in transforming parallel branches in a non-deterministic single branch (using a process that the author call *expansion*), which transforms the issue of connectedness of repeated operations into one or more issues of connectedness of sequences and of choices, which are then solved with the previously relative strategies.

Chapter 3

ChorTex: A Simple Choreography Modeling Language

This chapter is devoted to introducing ChorTex¹, the choreography modeling language adopted throughout this thesis. ChorTex is based on Chor [205], a process algebra for modeling choreographies that provides mechanisms for exception handling inspired by those found in WS-BPEL.

This chapter is structured as follows. The assumptions underpinning the design of ChorTex are outlined in Section 3.1. The syntax and operational semantic of ChorTex are presented in Section 3.2 and Section 3.3. Section 3.4 discusses the generation of Control Flow Graphs from ChorTex choreographies. In order to simplify algorithms and explanations in later parts of this thesis, for example by avoiding treating corner-cases that would not normally occur in real-world choreographies, the concept of well-formedness for ChorTex choreographies is formalized in Section 3.5. Finally, the details and rationale of the improvements of ChorTex over its progenitor Chor are discussed in Section 3.6.

3.1 Design Assumptions

The design assumptions that underpin ChorTex have very important implications on both the realizability definition and realizability analysis method presented in Chapter 4 as well as the remediation strategies presented in Chapter 5.

In Section 2.3.1.2 we have argued that the communication model is a fundamental aspect of any realizability definition. According to the taxonomy of communication models presented in [132], ChorTex assumes the following:

Design Assumption 1 (Asynchronous communication model). The message-based interactions between participants of a ChorTex choreography are carried out according to the following type of asynchronous messaging:

Sending strategy: non-blocking sending, always succeed;

Buffer: The participant implementations of ChorTex choreographies use queues with the following characteristics:

Ordering: ordered (First in, first out); recipients receive messages in the order they are sent by the sender

Capacity: unbounded

¹The name kind of picked itself. It is a choreography modeling language, hence the “Chor” part. It is also a text-based DSL, and in many things it is reminiscent of markup languages, hence the “Tex” part. Moreover, “ChorTex” is pronounced like “cortex,” which is brainy enough to irresistibly compel this author.

Quantity: multiple buffers, one per participant; buffers are specific to single enactments (messages of the same type sent during different enactments are queued up in different buffers)

For the realizability analysis method presented in Chapter 4, it is important to point out that, due to the asynchronous nature of message exchanges in ChorTex, the recipient of a message exchange will receive and consume the message at some point in the future after its delivery. There is no guarantee about the point in time in which the recipient will receive and consume the message. More specifically, there is no guarantee that multiple recipients of a message will consume it at the same time, nor in any particular order (i.e., a certain participant consuming a given message before or after another participant).

The assumed asynchronous communication model may sound “strong,” with sending of messages that is always successful and ordered queues. However, it is feasible in current state of the art of SOA and BPM by tapping the high-availability features of modern messaging services and policies like “exactly-once” that are offered by middleware facilities that implement technologies like WS-ReliableMessaging. The messaging services can guarantee us that once a message is dispatched, it will be eventually received without corruption by the intended recipients.

What the assumed communication model does not guarantee, however, is that the recipients will be able to understand in the scope of which message exchange of which enactment is a particular message been dispatched. As discussed in Section 2.3.4.4, very insidious realizability defects such as non-deterministic choice and race-choice issues arise when recipients can misinterpret the relation between received messages and enacted message exchanges. We sidestep these issues by means of the following two design assumptions:

Design Assumption 2 (Unequivocal enactments). Given a message, the recipient is able to correctly and unequivocally identify the enactment in the scope of which that message has been sent.

It is easy to fulfill the “Unequivocal enactments” Design Assumption with technical means. Consider, for example, the case of WS-BPEL. Multiple instances of the same WS-BPEL executable process can run concurrently at the same endpoint. Each of those instances is taking part to a different enactment. Upon the reception of a message, the orchestration engine decided to which instance it will be dispatched on the basis of the **correlation sets** defined in the WS-BPEL orchestration. Correlation sets are fundamentally pattern-matching rules that uniquely correlate messages with specific instances based on data such as order or transaction identifiers that are found in the messages themselves. Similar mechanisms are also built directly in specifications that deal with distributed transactions. For example, **coordination context identifiers** in WS-Coordination [97] and **conversationId** SOAP header supported by the Seam Framework² are designed exactly to achieve the same one-to-one correspondence between messages and enactments from the point of view of recipients. In ChorTex, we assume that messages contain *enactment identifiers* that allow recipients to correctly map messages to enactments. In ChorTex choreographies, the enactment identifier is generated by the first participant performing an action in the enactment (therefore “kick-starting” the enactment itself) and it is then propagated using message exchanges to the other participants. The correct propagation of enactment identifiers among participants is ensured by the realizability analysis method discussed in Section 4.6.

However, correctly mapping received messages to the enactments they belong to is generally not good enough. For example, another factor that contributes to causing non-deterministic and race-choice issues in Message Sequence Charts is that, even within the scope of one enactment, recipients may mistake which message exchange activity has generated a given message. Mistakes in associating messages with the activities that generated them leads to desynchronization between senders and recipients: senders know that the enactment has reached one precise state (the one in which the activity has been completed), while the recipients wrongly infer that another state has

²JBoss Seam Framework website: <http://www.seamframework.org/>

been reached (one in which a different activity has been completed). In ChorTex, we solve this potential issue “at the source” by means of the following assumption:

Design Assumption 3 (Unequivocal message exchanges). Given a message, the recipient is able to correctly and unequivocally identify the message exchange activity in the scope of which that message has been sent.

Similarly to the case of the “Unequivocal enactments” assumption, the “Unequivocal message exchanges” assumption is readily satisfied with technical means by combining the following two constraints:

1. Given a message, the recipient can always unequivocally identify its message type;
2. All message exchanges in a choreography must use different message types.

Figure 3.1 shows how these two constraints enable recipients of message to correctly identify which message exchange activity has generated a message they received. Through the message type identifier embedded in the message, the recipient knows to which message type the message belongs. And thanks to the one-to-one correspondence between message types and message exchange activities, the recipient can trace the message back to the activity that generated it. Both constraints are very simple to realize. The one-to-one correspondence between message types and activities can be statically verified on the choreography (see Section 3.5). In general, the capability of correctly identifying the message type of a message via static verification is potentially very complicated due to complexity and expressiveness of data languages like XML Schema Definition or the lack of integrated schema capabilities of others like JSON³ and MessagePack⁴. However, in our case the solution is straightforward: senders must include the corresponding message type identifiers as meta-data of the messages. In the scope of SOA technologies, this can be done, for example, using a dedicated SOAP header.

The “Unequivocal enactments” and “Unequivocal message exchanges” design assumptions stated so far focus on recipients of message exchanges and enable them to understand correctly which enactments and which activities are sources for messages they received. But how about the senders of the message exchanges? And, in particular, how are senders supposed to know *where* to send their messages, specifically in term of endpoints? ChorTex is fundamentally a process algebra. As discussed in Section 2.2.1.1, many process algebras offer constructs that realize *channel passing*, i.e., the possibility for participants to exchange information on the identity of other participants during the enactment, and therefore determine dynamically *who takes part in the enactment*. Features to the same end are also provided by choreography modeling languages not based on process algebras. For example, BPEL4Chor has the **participant sets** [83] and BPMN v2.0 Choreography has **multi-instance participants** (see Section 2.2.2.4). Channel passing, however, requires extra verification: before a participant can send a message to another, it must be necessarily the case that the first participant has acquired a reference (a.k.a. a *handle*) to the second, either by receiving said reference from another participant (the “passing” in channel passing) or by determining on its own the recipient in however way dictate by that participant’s internal logic.⁵

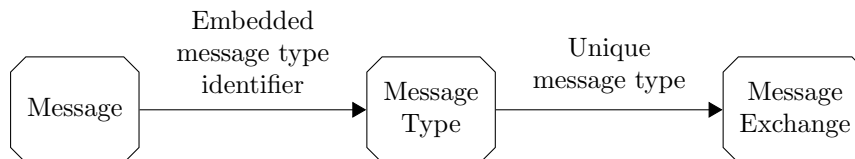


Figure 3.1: How recipients “trace back” messages to the message exchanges that generated them.

³JSON website: <http://www.json.org/>.

⁴MessagePack website: <http://msgpack.org/>.

⁵Which, stretching further the metaphor of choreography as inspired by dancing, fundamentally realizes a “dance invitation” from the sender to the recipient.

For reasons of brevity of this work⁶, it was decided *not* to support channel passing in this formalization of ChorTex and the associated realizability analysis method and remediation strategies. However, we sketch how to introduce and handle channel passing in ChorTex in the scope of future work (see Section 7.3). The following design assumption formalizes the lack of channel passing facilities in ChorTex:

Design Assumption 4 (Participants are known a-priori). Participants in a ChorTex choreography are uniquely identified by *name*. The name of a participant is sufficient information for other participants to send messages to its participant implementation.

That is, we assume that the *grounding* of a participant, i.e., technical information such as endpoints and accepted transport protocols, is known to all the other participants that must exchange messages with it *before* the enactment begins. The most obvious way of achieving this goal is to provide to each participant with, alongside the choreography, also the grounding of all the other participants they may interact with (which, it is worth pointing out, may not necessarily be *all* the participants in the choreography). Alternatively, this assumption can be satisfied through *service discovery*, i.e., the process of “locating the appropriate service for a given task, where appropriate has a user-specific definition” [77]. In this case, the name of the participant in the choreography could be a Uniform Resource Identifier (URI) that provides means of “looking up” the technical information of the relative participant implementation. A similar approach is often adopted in conjunction with ESB implementations to improve the decoupling between services and their clients [68, 183]. In particular, services are identified by name, and their technical information (including but not limited to endpoints) is retrieved at run-time by the service clients through the service discovery facilities integrated in the ESB.

3.2 Outlook and Syntax of ChorTex

The syntax of ChorTex is presented in Figure 3.2 by means of the Xtext DSL⁷. The syntax of the Xtext DSL is based on Backus-Naur Form (BNF) grammars, with the addition of the possibility of naming non-terminal symbols. For example, consider the following rule:

BlockActivity: ‘{’ ‘[’ name=ID ‘]’ activities += **Activity** (‘;’ activities += **Activity**)* ‘}’

The example above means that the non-terminal symbol **BlockActivity** has at its two ends open and closed brackets, namely the ‘{’ and ‘}’ literals, which surround an identifier, called “name,” enclosed in square brackets and one or more productions of the non-terminal symbol **Activity**, with every two contiguous productions separated by the literal ‘;’. The productions of the non-terminal symbol **Activity** are grouped under the name of “activities” using the += operator, which concatenates a terminal to a list of terminals. In Xtext’s grammar, the symbols +, ? and * represent cardinalities, meaning that the groups preceding them (groups are delimited by parentheses, like in the case of the additional productions of **Activity** in the example) have to appear at least one, zero or one, or any number of times, respectively.

A ChorTex choreography specifies a *body*, which represents the “normal flow” of activities, and *exception handlers* that specify the activities to be enacted in reaction of exceptions propagating from the body. If the body *successfully completes*, i.e., no exception is raised during the body’s enactment, the enactment of the choreography successfully completes as well. On the contrary, if the enactment of the body results in an exception of type *e* been thrown, the exception handlers of that choreography are matched against *e*. The *body* of an exception handler consists of the activities to be enacted when that exception handler is triggered. In ChorTex there are two types of exception handlers: *named* and *default*.

⁶And, more precisely, to curb the already alarming lack thereof.

⁷Xtext (www.eclipse.org/Xtext/) is a Model-Driven Architecture (MDA) framework based on Eclipse for specifying textual DSLs. A guide to the syntax of Xtext’s grammar can be found at the following page: <http://www.eclipse.org/Xtext/documentation/>

Non-Terminal Symbol	Rule
Choreography:	<code>'chor' ([name=ID])? (' body=Activity (' namedExceptionHandlers+=NamedExceptionHandler)* (' defaultExceptionHandler=DefaultExceptionHandler)? ');</code>
NamedExceptionHandler:	<code>exceptionType=ID ':' body=BlockActivity;</code>
DefaultExceptionHandler:	<code>'*' ':' body=BlockActivity;</code>
Activity:	<code>BasicActivity ComplexActivity;</code>
BasicActivity:	<code>SkipActivity MessageExchangeActivity OpaqueActivity ThrowActivity;</code>
ComplexActivity:	<code>BlockActivity ChoiceActivity IterationActivity ParallelActivity Choreography;</code>
SkipActivity:	<code>'skip' ([name=ID])?;</code>
MessageExchangeActivity:	<code>(' [name=ID])? sender=ID '→' messageType=ID 'to' recipients+=ID (' , recipients+=ID)*;</code>
ThrowActivity:	<code>'throw' ([name=ID])? exceptionType=ID;</code>
OpaqueActivity:	<code>'opaque' ([name=ID])? (' participants+=ID (' , participants+=ID)+ ');</code>
BlockActivity:	<code>(' [name=ID])? '{' activities+=Activity (' ; activities+=Activity)* '};</code>
ChoiceActivity:	<code>'choice' ([name=ID])? decisionMaker=ID 'either' branches+=BlockActivity ('or' branches+=BlockActivity)+;</code>
IterationActivity:	<code>'iteration' ([name=ID])? decisionMaker=ID 'do' body=BlockActivity;</code>
ParallelActivity:	<code>'parallel' ([name=ID])? 'do' branches+=BlockActivity ('and' branches+=BlockActivity)+;</code>

Figure 3.2: The syntax of ChorTex expressed using the Xtext DSL; the non-terminal symbols are typeset in bold.

A named exception handler catches exceptions of one single type. Unlike Java and similarly to WS-BPEL, exception types in ChorTex do not have type hierarchy. Instead, the match between the type of the propagating exception and the one declared by the named exception handler is *literal*, i.e., matching string-wise the names of the two exception types. Default exception handlers can handle any type of exception, and intercept any exception propagating from the body that is not otherwise caught by named exception handlers. In other words, named exception handlers have precedence on the default one when determining which will catch an exception. If no exception handler (neither named nor default) for an exception is found, the choreography terminates and propagates the exception to its parent activity (if any is specified). When a choreography is terminated, its body (or the currently running exception handler) is terminated “on cascade.” When an exception propagates outside the *root choreography*, i.e., the “outer” choreography and the only one not having a parent choreography, the entire enactment is terminated. If an exception handler matching the thrown exception is found, its body is enacted. If the body of the exception handler completes, i.e., the exception has been dealt with, the enactment of the choreography completes successfully. (However, a body that is terminated because of an exception propagating from it is not “resumed” after that the exception has been handled.) Otherwise, if the enactment of body of the exception handler results in another exception being thrown, the choreography terminates by propagating this last exception to its parent activity (if the choreography was nested into another), or terminating the enactment if the choreography is the root one.

When the asynchronous message exchange $p_s \rightarrow m$ to p_{r_1}, \dots, p_{r_n} is enacted, the participant p_s sends a message of type m to the participants p_{r_1}, \dots, p_{r_n} . The participant that dispatches the message is called *sender*, namely p_s in the previous example. The one or more participants that receive the message are its *recipients*.

The **skip** activity is the “empty” activity. The enactment of a skip activity involves no actions performed by the participants and it always completes successfully and instantaneously.⁸

The activity **opaque** (p_1, \dots, p_n) represents an unspecified part of the choreography that involves the participants p_1, \dots, p_n . That is, an opaque activity is a “free form” activity that, when enacted, allows its participants to engage in any amount and ordering of message exchanges. Opaque activities eventually complete successfully (for simplicity, it is not allowed for opaque activities to throw exceptions). The particular message exchanges to be performed and their order can be specified later in the modeling of the choreography or at run-time by the participants that partake that opaque activity in the fashion of ad-hoc modeling [5, 198]. Irrespective of which of the two options is adopted, the participants of the opaque activity agree on its completion. In other words, all participants are assumed know when the enactment of the opaque activity is completed. It is outside the scope of this work to specify how the participants achieve this. This provision is necessary for the soundness of the operational semantics of ChorTex. The actual mechanisms that the participants employ to achieve this agreement on the completion of the opaque activity is outside the scope of this work. For example, the participants might have an agreed-upon protocol that is enacted in place of the opaque activity.

The enactment of the activity **throw** e results in an exception of type e being thrown.

The block activity $\{ A_1, \dots, A_n \}$ denotes the sequential enactment of the activities A_1, \dots, A_n . The completion of the first activity triggers the enactment of the second, and so on until all activities have been completed. If the enactment of one activity results in an exception being thrown, the next activities (if any) are not enacted and the exception is propagated.

The construct **parallel do** A_1 **and** \dots **and** A_n specifies the concurrent enactment of the *branches* A_1, \dots, A_n . To simplify the later presentation and at no cost to the expressiveness of ChorTex, it is required that each branch is specified as a block; of course, however, a block representing a branch may very well contain just a single activity. A parallel activity completes successfully when all its branches have completed successfully. If the enactment of one of the branches results in an exception being thrown, the other branches that have not yet completed are immediately terminated, the parallel activity is itself terminated, and the exception is propagated

⁸While semantically void, skip activities are extremely useful, as they allow the specification of logic like “when an exception of type e is caught, do nothing.”

to the parallel activity's parent.

The construct **choice** p **either** A_1 **or** ... **or** A_n models the conditional choice (i.e., “if then else”) of which of the $A_1 \dots A_n$ *branches* to execute. Similarly to the case of parallel activities, the branches are blocks. The decision about which branch is enacted is taken *internally* by the participant p , which is said to be the *decision maker*. The decision may be taken based on any combination of internal information and information available to the decision maker in terms of the current status and history of the enactment. Since the decision is internal, the choice construct does not specify the criteria used by the decision maker for deciding which branch is executed. To the other participants, the outcome of the decision maker's decision is not immediately apparent. The participants other than the decision maker must understand which branch is enacted by observing what happens *after* the decision maker has taken the decision, e.g., by observing which message exchanges take place thereafter.

Finally, the iteration activity **iteration** p **do** A specifies the repeated enactment (i.e., the “while-do”) of the activity A , which is said to be the *body* of the iteration. Similarly to the choice construct, the decision whether or not to iterate the body is taken internally by the decision maker p . Each time the body is completed successfully, the decision maker has to decide again whether to perform a further iteration or not. (In other words, the amount of iterations of the body is not generally pre-determined, though decision-maker participant implementations may of course internally elect to do so.) If an exception propagates from the body, the iteration activity is terminated and the exception is propagated to its parent.

Figure 3.3 introduces the ChorTex choreography $chor_1$ that is adopted as running example in the remainder of this chapter. The body $chor_1$ *body* of the running example contains two nested choreographies, $chor_2$ and $chor_3$, that are composed sequentially. The nested choreography $chor_2$ has in its body the parallel execution of two branches, prl_1 *branch* $_1$ and prl_1 *branch* $_2$, each consisting of one message exchange. After both parallel branches are completed, the message mex_3 is carried out.

The choreography $chor_2$ contains no **throw** activities, and therefore it is guaranteed to complete successfully. (It should be noted that we are not considering runtime errors, e.g., programming errors affecting participant implementations, which do not concern realizability.) Nevertheless, it defines a default exception handler, the body of which contains only the **skip** activity $skip_1$, which is never going to be enacted and it is, fundamentally, “dead code.” (In Section 3.4 it is shown how to detect this type of occurrences.) After the successful completion of $chor_2$, $chor_3$ is enacted. The body of $chor_3$ consists of the choice activity $choice_1$. Depending on the outcome of the decision of p_1 , either an exception of type e_1 is thrown (branch $choice_1$) or the message exchange mex_4 takes place. If the exception is thrown, the body of the choreography $chor_3$ is terminated and the exception propagates to the parent choreography $chor_1$. The exception is caught by the named exception handler that catches exactly exceptions of type e_1 , and its body $chor_1$ *nh* is enacted. The body of the named exception handler results in the throwing of another exception, this time of type e_2 , and $chor_1$ is terminated. Since $chor_1$ has no parent choreography (i.e., it is the root choreography), the entire enactment is terminated. Similarly to the case of the default exception handler of $chor_2$ is never going to be enacted, so is the one of $chor_1$.

3.3 Formal Foundation of ChorTex

The indispensable role of formal foundation for choreography modeling languages has been underlined in Section 2.2.2. Therefore, the goal of this section is to provide a formal operational semantics for the constructs of ChorTex. The necessary groundwork is presented in Section 3.3.1, while the operational semantics of each construct is detailed in (Section 3.3.2).

3.3.1 Basic Definitions

During the enactment of a choreography, its participants perform *actions* such as the dispatching of a message or deciding which branch of a choice activity to enact.

```

1  chor [chor1] (
2    [chor1body] {
3      chor [chor2] (
4        [chor2body] {
5          parallel [prl1] do [prl1branch1] {
6            [mex1] p2 → m1 to p1
7          } and [prl1branch2] {
8            [mex2] p3 → m2 to p1, p2
9          };
10         [mex3] p3 → m3 to p1, p2
11       }
12       | *: [chor2dh] {
13         skip [skip1]
14       }
15     );
16     chor [chor3] (
17       [chor3body] {
18         choice [choice1] p1
19         either [choice1either] {
20           throw [throw1] e1
21         } or [choice1or] {
22           [mex4] p1 → m4 to p2, p3
23         }
24       }
25     )
26   }
27   | e1: [chor1nh] {
28     throw [throw2] e2
29   }
30   | *: [chor1dh] {
31     skip [skip2]
32   }
33 )

```

Figure 3.3: The running example of this chapter.

Definition 3.1 (Actions and Acting Participants). The participants that perform a certain action are its *acting participants*. Table 3.1 correlates the various types of actions that are performed in ChorTex choreographies with their acting participants.⁹ The function *actingParticipants*(*n*) returns the set of acting participants of an action as specified in Table 3.1.

Table 3.1 does not show the “nested” actions in the branches of **choice** and **iteration** activities; however, they can be straightforwardly extracted using recursion and a few rules to compose actions resulting from sequences activities in blocks and the activities defined in a choreography’s body and exception handlers.

Another thing worth noting is that, perhaps counter-intuitively, **throw** activities are not mapped to actions. The reason for this is that the exception handling mechanism in ChorTex is a sort of control-flow construct for specifying the interruption of the enactment of concurrently running activities and the triggering of others as a result. In fact, unlike programming languages like Java or orchestration languages like WS-BPEL, an exception thrown while enacting a ChorTex

⁹ is a weather symbol representing fog; it seems fitting, since “what happens” during the enactment of an opaque activity is visible neither in the enactment trace nor is modeled in the choreography.

Activity	Action	Acting Participants
$[mex] p_s \rightarrow m \text{ to } p_{r_1}, \dots, p_{r_n}$	$[mex] p_s \xrightarrow{m} p_{r_1}, \dots, p_{r_n}$	p_s
opaque $[o] (p_1, \dots, p_n)$	$[o] \overset{\text{ }}{\text{}}(p_1, \dots, p_n)$	p_1, \dots, p_n
choice $[c] p \text{ either } A_1 \text{ or } \dots \text{ or } A_n$	$[c] p \xrightarrow{?} \mathbf{x}, \mathbf{x} \in \{A_1, \dots, A_n\}$	p
iteration $[i] p \text{ do } A$	$[i] p \xrightarrow{\circ} \mathbf{x}, \mathbf{x} \in \{\top, \perp\}$	p

Table 3.1: The types of actions and corresponding acting participants.

choreography is not represented by a data-structure. In other words, the throwing of an exception represents a “jump” in the enactment of the choreography and in possible interruption of some of the activities that are currently been enacted.¹⁰

Definition 3.2 (Enactment Traces). An *enactment trace* $(\mathbf{a}_1, \dots, \mathbf{a}_n)$ is a sequence of temporally-ordered actions $\mathbf{a}_1, \dots, \mathbf{a}_n$ of types defined in Table 3.1 performed collectively by the participants of a choreography. The concatenation of enactment traces is performed through the operator \circ , which is defined as follows:

$$(\mathbf{a}_1, \dots, \mathbf{a}_m) \circ (\mathbf{a}'_1, \dots, \mathbf{a}'_n) = (\mathbf{a}_1, \dots, \mathbf{a}_m, \mathbf{a}'_1, \dots, \mathbf{a}'_n)$$

The concatenation of two traces requires a strict temporal order between them. In particular, all the actions specified by the first trace have been enacted before those of the second trace.

Not all possible enactment traces are *valid* for a certain choreography:

Definition 3.3 (Valid Enactment Traces). An enactment trace σ is *valid* for a choreography if the participants do not violate that choreography by performing in order the actions specified by σ .

The validity of an enactment trace is verified by simulating on the choreography the sequence of actions it specifies by applying the operational semantics rules specified in Figure 3.13 and Figure 3.14. (Thanks to the “Unequivocal message exchanges” design assumption, simulation is always possible due to the lack of non-determinism and ambiguity in playing out the enactment trace.) Notice that, if a choreography specifies iteration activities, there may possibly be infinitely many valid enactment traces.

The *initiating actions* of a choreography are those actions that, when performed by their acting participants, may “kick-start” enactments of that choreography. (It is important to differentiate between initiating and non-initiating actions because the two types are treated differently in terms of the realizability analysis.)

Definition 3.4 (Initiating Actions). An action is *initiating* if its execution by the acting participants may cause the initiation of an enactment.

Intuitively, the possible initiating actions of a choreography are all and only those that can appear as first in valid enactment traces. A choreography may have more than one initiating action. This is, for example, the case of the choreography shown in Figure 3.3, which has two initial activities, namely the message exchange actions that result from the message exchanges mex_1 and mex_2 . The “Unequivocal enactments” design assumption guarantees that the messages resulting from multiple initial actions are still “correctly grouped” in one enactment. In Section 4.6 we will show how this is handled in the scope of the proposed realizability analysis method.

¹⁰For the record, GOTO-like constructs are widely considered to be, to put it mildly, not a particularly good idea (see, e.g., [203]). As it happens, this author whole-heartedly agrees and considers them detrimental not only in programming languages, but in choreography modeling languages as well. In fact, the reason for including the presented, activity-interrupting exception handling mechanism is to make a case *against this type of constructs* from the point of view of facilitating the modeling of realizable choreographies, see Section 4.7.2.

As participants perform actions during an enactment, the “state” of the latter changes. The *enactment state* is defined in terms of the enactment trace, i.e., the “history” of the enactment up to that moment, the state of each of the activities specified by the choreography (see the activity lifecycles in Figure 3.4 through Figure 3.12), and the *enactment mode*, i.e., whether an exception is being propagated or the enactment is “proceeding normally.”

Definition 3.5 (Enactment State). The state χ the enactment of a choreography is a tuple:

$$\chi := \llbracket \delta, \sigma, \mu \rrbracket$$

The symbol δ denotes the *enactment environment*, which is modeled as a *key-value map* data-structure that associates all choreography’s activities (the keys) with their states (the values). The possible states of activities is specified later in Section 3.3.2. The fact that the activity A is in the state s is denoted by:

$$\delta[A] = s$$

The symbol σ represents the enactment trace. The symbol μ denotes the *enactment mode*. There are two possible enactment modes. The symbol \checkmark denotes the *normal mode*, i.e., when no exception is being propagated; the *exception mode*, i.e., the mode in which an exception of type e is being propagated is denoted by $\checkmark e$.

3.3.2 Operational Semantics of ChorTex Constructs

This section presents the operational semantics of ChorTex constructs in two ways: as State Diagrams (SDs) of ChorTex elements (Figure 3.4 through Figure 3.12) as well as structured operational semantics (Figure 3.13 and Figure 3.14). While semantically equivalent, these two different representations have different readability and intended use. On one hand, structured operational semantics has been widely adopted to describe the meaning of process algebras and its rules translate very well to implementations and testing thereof. On the other hand, SDs representing the lifecycle of ChorTex elements are easier to read and provide a perspective on the enactment of a ChorTex choreography from the perspective of single choreography elements, which should come far more intuitive to the reader familiar with workflows and service composition languages than structured operational semantics.

Often, approaches to structured operational semantics, the progress of the enactment is tracked by “rewriting” the choreography specification removing the activities that have already been completed. On the contrary, the operational semantics of ChorTex presented in Figure 3.13 and Figure 3.14 adopts a “state-based” style, representing the current progress of the enactment as a combination of the states of the single activities. The reason adopting this style is that the resulting operational semantics mirrors closely the lifecycles of ChorTex elements shown in the SDs; ideally, the two notations should complement each other in facilitating the understanding of the operational semantics of ChorTex constructs.

In Figure 3.13 and Figure 3.14, the symbols p denote participant identifiers, m message types and e exception types. The assignment of the state s to the activity A is denoted by:

$$\delta[A \setminus s]$$

The symbols that represent the states of the activities are the same adopted in the activities’ lifecycles depicted in Figure 3.4 through Figure 3.12.

Skip activity: A skip activity, the lifecycle of which is shown in Figure 3.4, is instantaneous (see Section 3.2). This is represented in Rule Skip, which can be read in natural language as follows: “as soon as the activity skip is initiated (i.e., its state changes to ⊠), it completes successfully (i.e., its state becomes ⊡).” Notice that skip activities can be enacted only when the enactment mode is normal (i.e., denoted by \checkmark). Moreover, since skip activities are instantaneous, when they are initiated can never be interrupted by exception propagation.

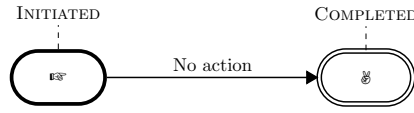


Figure 3.4: The enactment lifecycle of a skip activity.

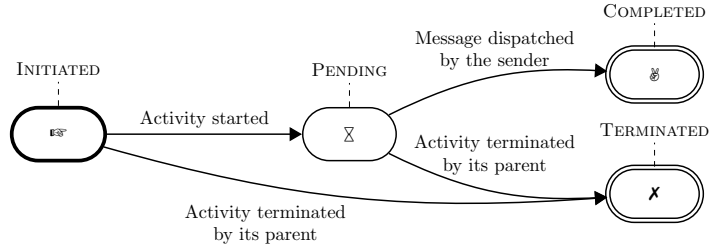


Figure 3.5: The enactment lifecycle of a message exchange activity.

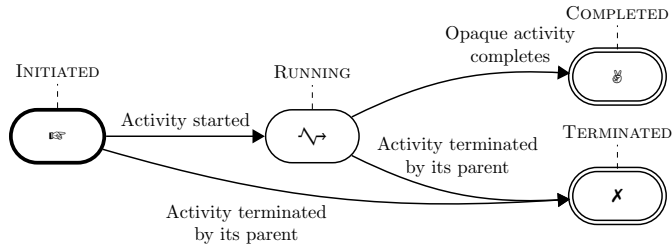


Figure 3.6: The enactment lifecycle of an opaque activity.

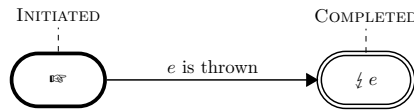


Figure 3.7: The enactment lifecycle of a throw activity.

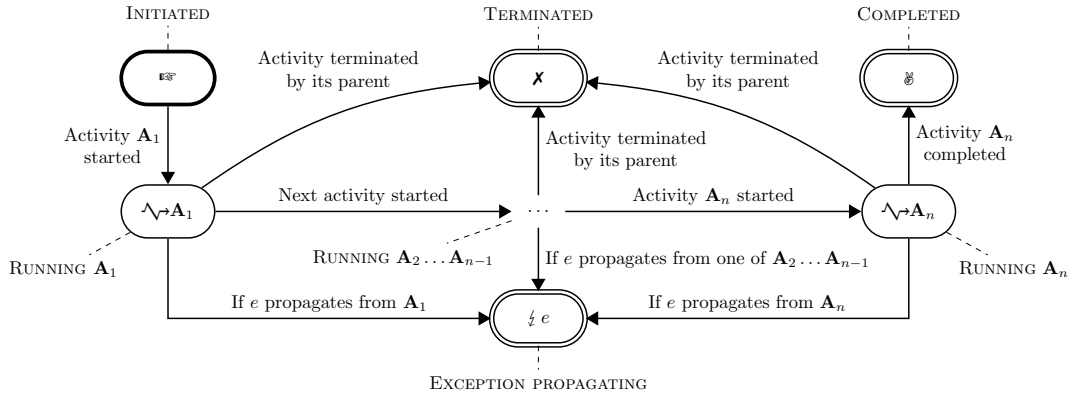


Figure 3.8: The enactment lifecycle of a block.

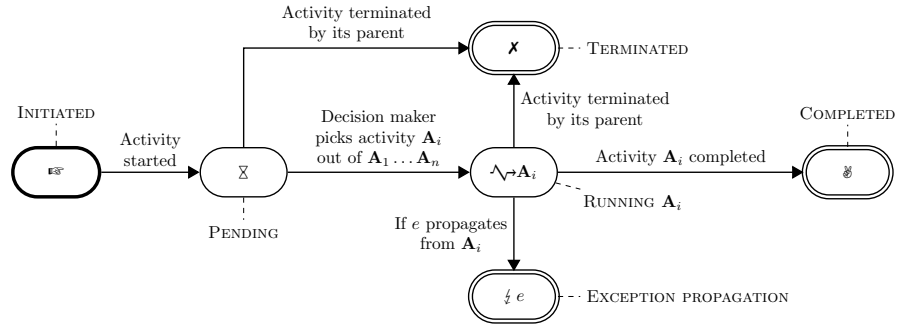


Figure 3.9: The enactment lifecycle of a choice activity.

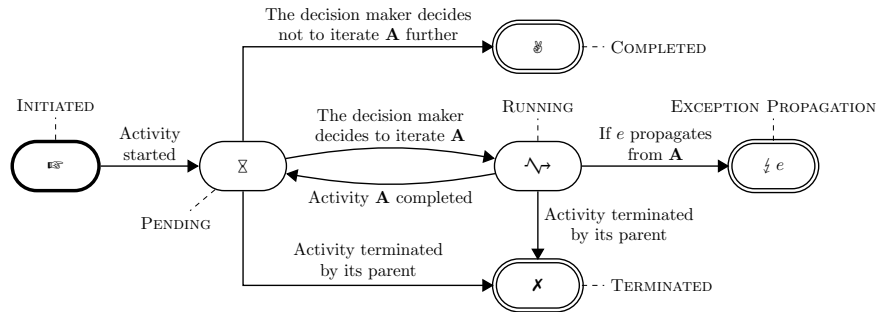


Figure 3.10: The enactment lifecycle of an iteration activity.

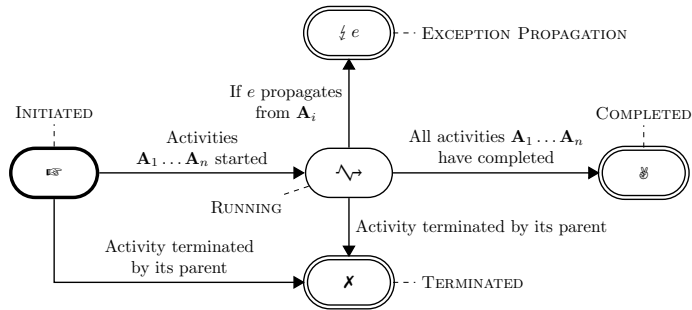


Figure 3.11: The enactment lifecycle of a parallel activity.

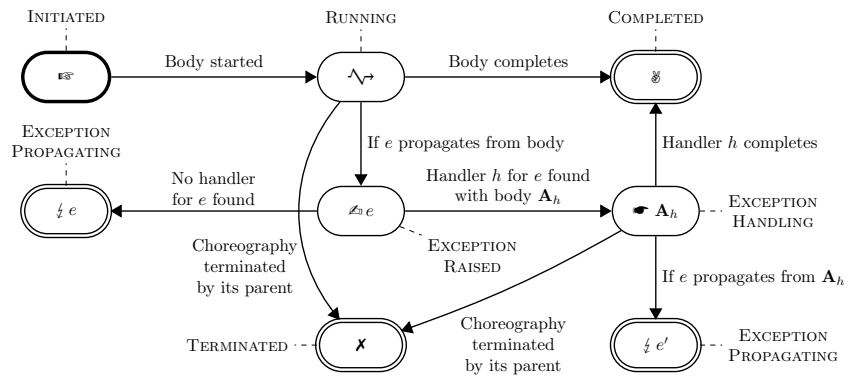


Figure 3.12: The enactment lifecycle of a choreography.

In the following rules, s stands for: $skip$

$$\frac{\delta[s] = \mathbb{E}^{\mathfrak{S}}}{\llbracket \delta, \sigma, \checkmark \rrbracket \rightarrow \llbracket \delta[s \setminus \mathfrak{E}], \sigma, \checkmark \rrbracket} \quad (\text{Skip})$$

In the following rules, mex stands for: $p_s \rightarrow m$ to p_{r_1}, \dots, p_{r_n}

$$\frac{\delta[mex] = \mathbb{E}^{\mathfrak{S}}}{\llbracket \delta, \sigma, \checkmark \rrbracket \rightarrow \llbracket \delta[mex \setminus \mathfrak{X}], \sigma, \checkmark \rrbracket} \quad (\text{Message Exchange Initiation})$$

$$\frac{\delta[mex] = \mathfrak{X} \wedge p_s \text{ dispatches } m \text{ to } p_{r_1}, \dots, p_{r_n}}{\llbracket \delta, \sigma, \checkmark \rrbracket \rightarrow \llbracket \delta[mex \setminus \mathfrak{E}], \sigma \circ ([mex] p_s \xrightarrow{m} p_{r_1}, \dots, p_{r_n}), \checkmark \rrbracket} \quad (\text{Message Exchange Completion})$$

In the following rules, o stands for: $opaque(p_1, \dots, p_n)$

$$\frac{\delta[o] = \mathbb{E}^{\mathfrak{S}} \wedge p_1, \dots, p_n \text{ start enacting } o}{\llbracket \delta, \sigma, \checkmark \rrbracket \rightarrow \llbracket \delta[o \setminus \mathfrak{A}], \sigma, \checkmark \rrbracket} \quad (\text{Opaque Initiation})$$

$$\frac{\delta[o] = \mathfrak{A} \wedge p_1, \dots, p_n \text{ complete enacting } o}{\llbracket \delta, \sigma, \checkmark \rrbracket \rightarrow \llbracket \delta[o \setminus \mathfrak{E}], \sigma \circ ([o] \mathfrak{A} (p_1, \dots, p_n)), \checkmark \rrbracket} \quad (\text{Opaque Completion})$$

In the following rules, t stands for: $throw e$

$$\frac{\delta[t] = \mathbb{E}^{\mathfrak{S}}}{\llbracket \delta, \sigma, \checkmark \rrbracket \rightarrow \llbracket \delta[t \setminus \mathfrak{E}], \sigma, \mathfrak{E} e \rrbracket} \quad (\text{Throw})$$

In the following rules, b stands for: $\{A_1; A_2; \dots; A_n\}$

$$\frac{\delta[b] = \mathbb{E}^{\mathfrak{S}}}{\llbracket \delta, \sigma, \checkmark \rrbracket \rightarrow \llbracket \delta[b \setminus \mathfrak{A} A_1][A_1 \setminus \mathbb{E}^{\mathfrak{S}}], \sigma, \checkmark \rrbracket} \quad (\text{Block Initiation})$$

$$\frac{\delta[b] = \mathfrak{A} A_i \wedge \delta[A_i] = \mathfrak{E} \wedge i < n}{\llbracket \delta, \sigma, \checkmark \rrbracket \rightarrow \llbracket \delta[b \setminus \mathfrak{A} A_{i+1}][A_{i+1} \setminus \mathbb{E}^{\mathfrak{S}}], \sigma, \checkmark \rrbracket} \quad (\text{Block Next Activity Initiation})$$

$$\frac{\delta[b] = \mathfrak{A} A_n \wedge \delta[A_n] = \mathfrak{E}}{\llbracket \delta, \sigma, \checkmark \rrbracket \rightarrow \llbracket \delta[b \setminus \mathfrak{E}], \sigma, \checkmark \rrbracket} \quad (\text{Block Completion})$$

$$\frac{\delta[b] = \mathfrak{A} A_i \wedge \delta[A_i] = \mathfrak{E} e}{\llbracket \delta, \sigma, \mathfrak{E} e \rrbracket \rightarrow \llbracket \delta[b \setminus \mathfrak{E}], \sigma, \mathfrak{E} e \rrbracket} \quad (\text{Block Exception})$$

$$\frac{\delta[b] = \mathfrak{X}}{\llbracket \delta, \sigma, \mathfrak{E} e \rrbracket \rightarrow \llbracket \delta[A_1 \setminus \mathfrak{X}] \dots [A_n \setminus \mathfrak{X}], \sigma, \mathfrak{E} e \rrbracket} \quad (\text{Block Termination})$$

In the following rules, c stands for: $choice p$ either A_1 or \dots or A_n

$$\frac{\delta[c] = \mathbb{E}^{\mathfrak{S}}}{\llbracket \delta, \sigma, \checkmark \rrbracket \rightarrow \llbracket \delta[c \setminus \mathfrak{X}], \sigma, \checkmark \rrbracket} \quad (\text{Choice Initiation})$$

$$\frac{\delta[c] = \mathfrak{X} \wedge p \text{ picks } A_i \text{ out of } A_1, \dots, A_n}{\llbracket \delta, \sigma, \checkmark \rrbracket \rightarrow \llbracket \delta[c \setminus \mathfrak{A} A_i][A_i \setminus \mathbb{E}^{\mathfrak{S}}], \sigma \circ ([c] p \xrightarrow{?} A_i), \checkmark \rrbracket} \quad (\text{Choice Decision})$$

$$\frac{\delta[c] = \mathfrak{A} A_i \wedge \delta[A_{i \in [1, n]}] = \mathfrak{E}}{\llbracket \delta, \sigma, \checkmark \rrbracket \rightarrow \llbracket \delta[c \setminus \mathfrak{E}], \sigma, \checkmark \rrbracket} \quad (\text{Choice Completion})$$

$$\frac{\delta[c] = \mathfrak{A} A_i \wedge \delta[A_{i \in [1, n]}] = \mathfrak{E} e}{\llbracket \delta, \sigma, \mathfrak{E} e \rrbracket \rightarrow \llbracket \delta[c \setminus \mathfrak{E}], \sigma, \mathfrak{E} e \rrbracket} \quad (\text{Choice Exception})$$

$$\frac{\delta[c] = \mathfrak{X} \wedge \delta[A_{i \in [1, n]}] \neq \mathfrak{E}}{\llbracket \delta, \sigma, \mathfrak{E} e \rrbracket \rightarrow \llbracket \delta[A_i \setminus \mathfrak{X}], \sigma, \mathfrak{E} e \rrbracket} \quad (\text{Choice Termination})$$

Figure 3.13: The structured operational semantics of ChorTex (Part 1).

In the following rules, i stands for: iteration p do A

$$\frac{\delta[i] = \mathbb{E}}{\llbracket \delta, \sigma, \checkmark \rrbracket \rightarrow \llbracket \delta[i \setminus \mathbb{X}], \sigma, \checkmark \rrbracket} \quad \text{(Iteration Initiation)}$$

$$\frac{\delta[i] = \mathbb{X} \wedge p \text{ internally decides to iterate } A}{\llbracket \delta, \sigma, \checkmark \rrbracket \rightarrow \llbracket \delta[i \setminus \mathbb{A}] [A \setminus \mathbb{E}], \sigma \circ ([i] p \overset{\circ}{\mapsto} \top), \checkmark \rrbracket} \quad \text{(Iteration Decision True)}$$

$$\frac{\delta[i] = \mathbb{X} \wedge p \text{ internally decides not to iterate } A}{\llbracket \delta, \sigma, \checkmark \rrbracket \rightarrow \llbracket \delta[i \setminus \mathbb{B}], \sigma \circ ([i] p \overset{\circ}{\mapsto} \perp), \checkmark \rrbracket} \quad \text{(Iteration Decision False)}$$

$$\frac{\delta[i] = \mathbb{A} \wedge \delta[A] = \mathbb{B}}{\llbracket \delta, \sigma, \checkmark \rrbracket \rightarrow \llbracket \delta[i \setminus \mathbb{X}], \sigma, \checkmark \rrbracket} \quad \text{(Iteration Body Completion)}$$

$$\frac{\delta[i] = \mathbb{X} \wedge \delta[A] \neq \mathbb{B}}{\llbracket \delta, \sigma, \zeta e \rrbracket \rightarrow \llbracket \delta[A \setminus \mathbb{X}], \sigma, \zeta e \rrbracket} \quad \text{(Iteration Termination)}$$

In the following rules, prl stands for: parallel do A_1 and \dots and A_n

$$\frac{\delta[prl] = \mathbb{E}}{\llbracket \delta, \sigma, \checkmark \rrbracket \rightarrow \llbracket \delta[prl \setminus \mathbb{A}] [A_1 \setminus \mathbb{E}] \dots [A_n \setminus \mathbb{E}], \sigma, \checkmark \rrbracket} \quad \text{(Parallel Initiation)}$$

$$\frac{\delta[prl] = \mathbb{A} \wedge \forall i \in [1, n] : \delta[A_i] = \mathbb{B}}{\llbracket \delta, \sigma, \checkmark \rrbracket \rightarrow \llbracket \delta[prl \setminus \mathbb{B}], \sigma, \checkmark \rrbracket} \quad \text{(Parallel Completion)}$$

$$\frac{\delta[prl] = \mathbb{A} \wedge \delta[A_{i \in [1, n]}] = \zeta e \wedge \{A_j \in [1, n] \mid \delta[A_j] \neq \mathbb{B}\} = \{A'_1, \dots, A'_m\}}{\llbracket \delta, \sigma, \zeta e \rrbracket \rightarrow \llbracket \delta[prl \setminus \zeta e] [A'_1 \setminus \mathbb{X}] \dots [A'_m \setminus \mathbb{X}], \sigma, \zeta e \rrbracket} \quad \text{(Parallel Exception)}$$

$$\frac{\delta[prl] = \mathbb{X} \wedge \{A_i \in [1, n] \mid \delta[A_i] \neq \mathbb{B}\} = \{A'_1, \dots, A'_m\}}{\llbracket \delta, \sigma, \zeta e \rrbracket \rightarrow \llbracket \delta[A'_1 \setminus \mathbb{X}] \dots \delta[A'_m \setminus \mathbb{X}], \sigma, \zeta e \rrbracket} \quad \text{(Parallel Termination)}$$

In the following rules, $chor$ stands for: chor ($A \mid e_1 : A_1 \mid \dots \mid e_n : A_n \mid * : A_*$)

$$\frac{\delta[chor] = \mathbb{E}}{\llbracket \delta, \sigma, \checkmark \rrbracket \rightarrow \llbracket \delta[chor \setminus \mathbb{A}] [A \setminus \mathbb{E}], \sigma, \checkmark \rrbracket} \quad \text{(Choreography Initiation)}$$

$$\frac{\delta[chor] = \mathbb{A} \wedge \delta[A] = \mathbb{B}}{\llbracket \delta, \sigma, \checkmark \rrbracket \rightarrow \llbracket \delta[chor \setminus \mathbb{B}], \sigma, \checkmark \rrbracket} \quad \text{(Choreography Completion)}$$

$$\frac{\delta[chor] = \mathbb{A} \wedge \delta[A] = \zeta e}{\llbracket \delta, \sigma, \zeta e \rrbracket \rightarrow \llbracket \delta[chor \setminus \zeta e], \sigma, \checkmark \rrbracket} \quad \text{(Choreography Exception 1)}$$

$$\frac{\delta[chor] = \mathbb{A} e \wedge \exists i \in [1, n] : e_i = e \wedge A_i \text{ is body of the named exception handler for } e_i}{\llbracket \delta, \sigma, \zeta e \rrbracket \rightarrow \llbracket \delta[chor \setminus \blacktriangleright A_i] [A_i \setminus \mathbb{E}], \sigma, \checkmark \rrbracket} \quad \text{(Choreography Exception 2)}$$

$$\frac{\delta[chor] = \mathbb{A} e \wedge \nexists i \in [1, n] : e_i = e \wedge A_* \text{ is body of the default exception handler}}{\llbracket \delta, \sigma, \zeta e \rrbracket \rightarrow \llbracket \delta[chor \setminus \blacktriangleright A_*] [A_* \setminus \mathbb{E}], \sigma, \checkmark \rrbracket} \quad \text{(Choreography Exception 3)}$$

$$\frac{\delta[chor] = \mathbb{A} e \wedge \nexists i \in [1, n] : e_i = e \wedge \text{no default exception handler specified}}{\llbracket \delta, \sigma, \zeta e \rrbracket \rightarrow \llbracket \delta[chor \setminus \zeta e], \sigma, \zeta e \rrbracket} \quad \text{(Choreography Exception 4)}$$

$$\frac{\delta[chor] = \blacktriangleright A_h \wedge \delta[A_h] = \mathbb{B}}{\llbracket \delta, \sigma, \checkmark \rrbracket \rightarrow \llbracket \delta[chor \setminus \mathbb{B}], \sigma, \checkmark \rrbracket} \quad \text{(Choreography Exception Handling 1)}$$

$$\frac{\delta[chor] = \blacktriangleright A_h \wedge \delta[A_h] = \zeta e}{\llbracket \delta, \sigma, \zeta e \rrbracket \rightarrow \llbracket \delta[chor \setminus \zeta e], \sigma, \zeta e \rrbracket} \quad \text{(Choreography Exception Handling 2)}$$

$$\frac{\delta[chor] = \mathbb{X} \wedge \delta[A] = \mathbb{A}}{\llbracket \delta, \sigma, \zeta e \rrbracket \rightarrow \llbracket \delta[A \setminus \mathbb{X}], \sigma, \zeta e \rrbracket} \quad \text{(Choreography Termination 1)}$$

$$\frac{\delta[chor] = \mathbb{X} \wedge \delta[A_{i \in [1, n]}] = \mathbb{A}}{\llbracket \delta, \sigma, \zeta e \rrbracket \rightarrow \llbracket \delta[A_i \setminus \mathbb{X}], \sigma, \zeta e \rrbracket} \quad \text{(Choreography Termination 2)}$$

$$\frac{\delta[chor] = \mathbb{X} \wedge \delta[A_*] = \mathbb{A}}{\llbracket \delta, \sigma, \zeta e \rrbracket \rightarrow \llbracket \delta[A_* \setminus \mathbb{X}], \sigma, \zeta e \rrbracket} \quad \text{(Choreography Termination 3)}$$

Figure 3.14: The structured operational semantics of ChorTex (Part 2).

Message exchange activity: The lifecycle of message exchange activities is shown in Figure 3.5. When its state is INITIATED and the enactment mode is “normal,” the message exchange activity enters the state PENDING, denoted by the hourglass-like symbol \boxtimes (Rule Message Exchange Initiation). After a message exchange activity enters the state PENDING, its completion is not necessarily instantaneous. Instead, the message exchange becomes *enactable*, meaning that, if the sender dispatches the message to the recipients, the choreography is not violated. In other words, the sender of a message exchange can delay the dispatch of the message indefinitely, thereby “stalling” the enactment of the message exchange activity. There are no restrictions on how long can the sender delay the completion of the message exchange. In normal enactment mode, the message exchange activity performs the transition from the PENDING to the COMPLETED state when the sender dispatches the message to the recipients (Rule Message Exchange Completion). Notice that, since Rule Message Exchange Completion assumes the enactment mode to be normal, participants that dispatch messages when the enactment state in exception mode are violating the choreography.

Since message exchange activities are not instantaneous, they may be terminated while they are in the state PENDING. When a message exchange activity is terminated, its state changes to TERMINATED, denoted by \boxtimes . Notice that there is no rule that represents in the specific the termination of a message exchange activity. Instead, the termination of a message exchange activity occurs while executing a rule that processes the termination of the complex activity that is parent to that message exchange activity. In particular, the rules that, when executed, can result in the termination of a nested message exchange activity are those for the termination of choreographies (Rule Choreography Termination 1, Rule Choreography Termination 2 and Rule Choreography Termination 3), blocks (Rule Block Termination), choice activities (Rule Choice Termination), iteration activities (Rule Iteration Termination) and parallel activities (Rule Parallel Termination).

Opaque activity: An opaque activity changes its state from INITIATED to RUNNING when its participants start enacting it (Rule Opaque Initiation). The participants can begin the enactment of an opaque activity only when the enactment mode is normal (Rule Opaque Initiation). When the participants complete an opaque activity, the latter changes its state to COMPLETED (Rule Opaque Completion). The participants can complete the enactment of an opaque activity only when the enactment mode is normal (Rule Opaque Completion). No restrictions apply to the amount of time that it takes to the participants to complete an opaque activity. Thus, opaque activities are not instantaneous, and can be terminated. Similarly to the case of message exchange activities, there is no rule specific to the termination of opaque activities. Instead, the termination of an opaque activity – i.e., its state changing to TERMINATED – occurs while executing a rule that processes the termination of the complex activity that is parent to that opaque activity.

Throw activity: Throw activities, like skip ones, are instantaneous: they complete as soon as they are initiated (Rule Throw). The execution of a throw e activity causes the enactment mode to change from normal to exception (denoted by $\downarrow e$).

Block: A block can be initiated only when the enactment mode is normal; when a block is initiated, its first nested activity is also initiated “on cascade” (Rule Block Initiation) In normal enactment mode, the completion of a nested activity that is not the last in the block triggers the initiation of the following one (Rule Block Next Activity Initiation). In normal enactment mode, the completion of the last nested activity results in the completion of the block (Rule Block Completion). If the enactment of any nested activities of a block results in the propagation of an exception, the block is terminated and the exception propagates further to the parent activity of the block (Rule Block Exception). Finally, when a block is terminated by its parent activity, the nested activity currently running and all the subsequent nested activities are also terminated (Rule Block Termination).¹¹ The COMPLETED states of the previously completed nested activities

¹¹Terminating not only the currently running nested activity, but also the ones following it has the goal of preventing unforeseen behaviors in case of later iterations of previously terminated blocks, which can happen in the

are not affected by the termination (Rule Block Termination). Notice that the termination of any activity, blocks included, is fundamentally the result of an exception been thrown and not yet handled. Therefore, the termination of a block can occur only when the enactment is in exception mode.

Choice activity: Choice activities, once initiated, enter the state PENDING (Rule Choice Initiation), which represents the “waiting” for the internal decision performed by the decision maker with respect to which of the branches is to be enacted. When the decision maker selects the branch \mathbf{A}_i to enact, the block that constitutes that branch is initiated (Rule Choice Decision) and the choice activity enters the state RUNNING \mathbf{A}_i . When the branch selected by the decision maker completes, so does the choice activity (Rule Choice Completion). If the enactment of the selected branch results in the propagation of an exception, the state of the choice activity becomes EXCEPTION PROPAGATION, and the exception is propagated to its parent (Rule Choice Exception). A choice activity can be terminated by its parent while the decision is pending, or when the selected branch has not yet completed; the termination of the choice cause the termination “on cascade” of the branches being enacted (Rule Choice Termination).

Iteration activity: Similarly to the case of choice activities, the enactment of an iteration activity begins with the transition of its state from INITIATED to PENDING (Rule Iteration Initiation), which represents the fact that the decision maker has not yet taken its decision on whether to execute once more the iteration’s body. The iteration activity completes if, while it is in the state PENDING, the decision maker decides not to enact the body further (Rule Iteration Decision False). Of course, it is possible that the decision maker decides the very first time against enacting the body, so it can be the case that the body of an iteration actually does not get enacted at all in enactments. Otherwise, the iteration activity enters the state RUNNING and the body is initiated (Rule Iteration Decision True). When the body completes, the iteration activity returns to the state PENDING, so that its decision maker can evaluate once more whether to iterate the body further (Rule Iteration Body Completion). If the enactment of the body results in the propagation of an exception, the iteration activity enters the state EXCEPTION PROPAGATION and propagates the exception to its parent. If the iteration is terminated by its parent and its body is not yet completed, the body is also terminated “on cascade” (Rule Iteration Termination).

Parallel activity: When a parallel activity is started, its state changes from INITIATED to RUNNING and all its branches are initiated (Rule Parallel Initiation). A parallel activity completes when all its branches are completed (Rule Parallel Completion).

If the enactment of one the branches results in the propagation of an exception, the parallel activity enters the state EXCEPTION PROPAGATION, the other branches that have not yet been completed are terminated, and the exception is propagated to the parent of the parallel activity (Rule Parallel Exception). If the parallel is terminated by its parent, all the branches that have not yet completed are terminated as well (Rule Parallel Termination).

Choreography & exception handling: When the enactment of a choreography begins, its state transitions from INITIATED to RUNNING, triggering the initiation of the choreography’s body (Rule Choreography Initiation). If the body completes, the enactment of the choreography completes (Rule Choreography Completion).

If the enactment of the body results in the propagation of an exception, the state of the choreography changes from RUNNING to EXCEPTION RAISED (Rule Choreography Exception 1). When

case of choreographies nested inside iteration activities. Since there are no rules that allow a TERMINATED state to transition to something else, the only way those activities can perform further state transitions is if they first go through a “reset” that moves them back to the INITIATED state, which takes place in Rule Block Initiation and Rule Block Next Activity Initiation. In this author’s opinion, it is probably possible to prove that with the current operational semantics rules, such a “bug” cannot occur, but a bit of “defensive programming” cannot hurt.

the choreography enters the state `EXCEPTION RAISED`, the exception handlers defined by the choreography are matched against the type of the exception. If a named exception handler matches the type of the propagating exception, the state of the choreography changes to `EXCEPTION HANDLING` and the body of that named exception handler is initiated (Rule Choreography Exception 2). If no matching named exception handler is found, but the choreography defines a default exception handler, the latter’s body is initiated (Rule Choreography Exception 3) and the state of the choreography changes to `EXCEPTION HANDLING`. If neither matching named exception handler is found nor the choreography defines an default exception handler (denoted by $A_* = \epsilon$ in Rule Choreography Exception 4), the choreography is terminated and the exception is propagated to its parent. If the choreography is the root choreography, and thus has no parent, the enactment itself is terminated.

If an exception handler matching the type of the propagating exception is found, i.e., the choreography is in the state `EXCEPTION HANDLING`, and the body of that exception handler completes, then the choreography completes as well (Rule Choreography Exception Handling 1). Otherwise, if the enactment of the exception handler’s body results in the propagation of another exception of type e' , the choreography’s state transitions to `EXCEPTION PROPAGATION` (denoted by $\zeta e'$), and the exception is propagated to the choreography’s parent (Rule Choreography Exception Handling 2). That is, exceptions thrown by the body of an exception handler cannot be caught by other exception handlers of the same choreography. Similarly to the case when no matching handler for an exception can be found, if the choreography is the root choreography, the enactment is terminated. The termination of a choreography can occur when its body or that of one of the exception handlers is been enacted (Rule Choreography Termination 1, Rule Choreography Termination 2 and Rule Choreography Termination 3), which is immediately terminated.

3.4 From ChorTex Choreographies to Control Flow Graphs

Some aspects of the well-formedness of ChorTex choreographies discussed later in Section 3.5 as well as the bulk of the realizability analysis presented in Chapter 4 build on top of the flow-analysis framework (see, e.g., [11, 6]). The flow-analysis framework is used to study *static properties* of computer programs, that is, properties reaching definitions and data dependences that can be inferred from the structure of the programs at design/compilation-time [6].

The flow-analysis framework is centered on the concept of Control Flow Graph (CFG). The Control Flow Graph of a program is a directed graph in which each node represents one of the program’s instructions. An edge connecting two nodes represents a control dependency between the respective instructions, meaning that the instruction represented by the *source* node of the edge is executed before the one represented by the *target* node. The same concepts are applied to ChorTex constructs in the remainder of this section. In particular, Section 3.4.1 explains how to construct Control Flow Graphs from ChorTex choreographies, while Section 3.4.2 correlates the Control Flow Graphs so constructed and the operational semantics of ChorTex presented in Section 3.3.

3.4.1 Construction of Control Flow Graphs of ChorTex Choreographies

We adapt to ChorTex choreographies the approach proposed by [189] for constructing Control Flow Graphs of Java programs. Table 3.2 and Table 3.3 show how to construct Control Flow Graphs of ChorTex choreographies by mapping every choreography activity to a Control Flow Graph sub-graph. The Control Flow Graph nodes with the grey background in Table 3.2 and Table 3.3 are “place-holders” for the sub-graphs of the respective activities. The compositionality of the mapping rules requires a mechanism for dealing with edges that connect nodes resulting from different activities. This is done by labeling edges with conditions like “to first(**A**),” which means that the target node of the edge is the one labelled as “first” in the sub-graph resulting from the activity **A**. Similarly, the condition “from last(**A**)” means that the source of the labelled edge is the node labelled as “last” in the sub-graph resulting from the activity **A**. Due to the definition

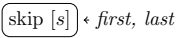
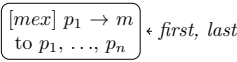
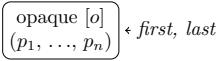
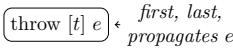
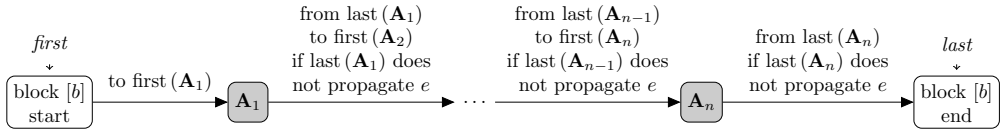
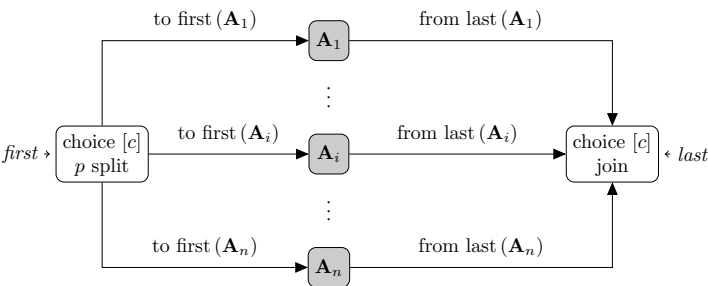
ChorTex Activity	Corresponding Control Flow Graph
skip [s]	
[<i>max</i>] $p_1 \rightarrow m$ to p_1, \dots, p_n	
opaque [o] (p_1, \dots, p_n)	
throw [t] e	
[b] { $\mathbf{A}_1 ; \dots ; \mathbf{A}_n$ }	
choice [c] p either \mathbf{A}_1 or \dots or \mathbf{A}_n	

Table 3.2: The mapping from ChorTex activities to Control Flow Graphs (Part 1).

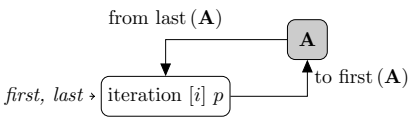
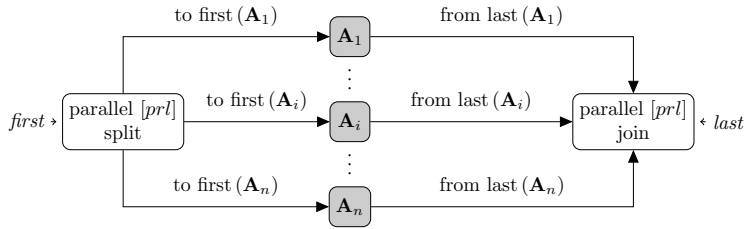
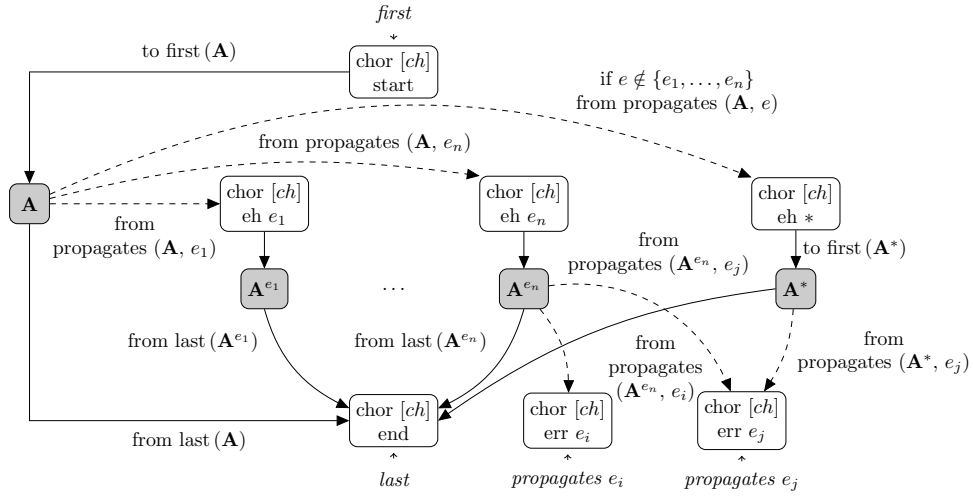
ChorTex Activity	Corresponding Control Flow Graph
iteration $[i]$ p do \mathbf{A}	
parallel $[prl]$ do \mathbf{A}_1 and ... and \mathbf{A}_n	
$\text{chor } [ch] (\mathbf{A} \mid e_1 : \mathbf{A}^{e_1} \mid \dots \mid e_n : \mathbf{A}^{e_n} \mid * : \mathbf{A}^*)$	

Table 3.3: The mapping from ChorTex activities to Control Flow Graphs (Part 2).

of “first” and “last” in Table 3.2 and Table 3.3, each Control Flow Graph sub-graph has precisely one “first” and one “last” node.

Each node of the Control Flow Graph generated from a ChorTex choreography represents the *firing* of exactly one event such as the beginning or completion of an activity. Consequently, the control-flow edges represent the order in which the firing of events may occur during enactments of the choreography. Intuitively, a control-flow edge connecting the two nodes n and n' , respectively representing the firing of the events e and e' , means that e' can be fired only after that e was fired. A more precise interpretation of the ordering of events is presented later in Section 3.4.2, as it requires an understanding of how the enacting of activities relates to the firing of events.

The nodes `block [b] start` and `block [b] end` that are generated from the block activity named b represent the beginning and completion of the latter, respectively. Similarly, the nodes `choice [c] p split` and `choice [c] join` represent the beginning and completion of the choice activity c , respectively.¹² The node `iteration [i] p` represents at the same time beginning, completion and the taking of the decision by p of the iteration activity i .¹³ The nodes `parallel [p] split` and `parallel [p] join` represent the beginning and completion of the enactment of the parallel activity p .

The enactment states of a choreography ch are the beginning (`chor [ch] start`), completion (`chor [ch] end`), invocation of the default exception handler (`chor [ch] eh *`) or named exception handler for the exception type e (`chor [ch] eh e`), and the propagation outside the choreography of an exception of type e (`chor [ch] err e`).

Dealing with exceptions requires special care. Nodes that represents the throwing or propagation of an exception e to parent activities are labelled “propagates e ”. Consider for example the block $\{ \mathbf{A}_1; \mathbf{A}_2 \}$. If \mathbf{A}_1 is a `throw e` activity, \mathbf{A}_2 will never be enacted; therefore the corresponding sub-graph in the Control Flow Graph must not have a control-flow edge connecting \mathbf{A}_1 with \mathbf{A}_2 . For the sake of understandability, the *exception control-flow* edges, i.e., the control-flow edges that represent the propagation of exceptions, are depicted dashed in Table 3.2 and Table 3.3. It should be noted that only the rule for creating Control Flow Graph sub-graphs of choreographies specify exception control-flow edges. The reason is that the “wiring” of exception propagating and handling is done only at the level of choreographies.

The running example in Figure 3.3 is a choreography that contains nested choreographies and presents non-trivial propagation of exceptions. The exception of type e_1 that can be thrown during the enactment of $chor_3$ is caught by the named exception handler for e_1 specified in $chor_1$, which in turn throws an exception of type e_2 that terminates the choreography. The exception propagation is evident in Figure 3.15, which shows the Control Flow Graph resulting from the choreography in Figure 3.3. It is interesting to notice that there are some nodes, such as `chor [chor1] eh *`, `skip [skip2]`, `chor [chor2] eh *` and `skip [skip1]`, that are not reachable – in terms of “pure” graph traversing – from the start node of the Control Flow Graph, i.e., the “first” node of the sub-graph generated from the root choreography. In Figure 3.15, the nodes not reachable from the start node are painted with reduced opacity (i.e., they look “more transparent” than others), as well as the control-flow edges that originate from and/or target them. Specifically, `block [choice1either] end` is not reachable because the last activity of its block is a *propagate* e_1 node, namely a `throw` node, and thus there is no edge connecting the `throw` and `block [choice1either] end` nodes. The other unreachable nodes, instead, represent exception handlers that are never triggered.

¹²It would be possible to map choice activities without resorting to join nodes. For example, we could connect the last nodes of the sub-graphs generated by the branches with the first node of the activity after the choice. However, having join nodes greatly simplifies the specification of the mapping rules and the specification of the realizability analysis method presented in Chapter 4.

¹³It would also be possible to represent separately these events with distinct nodes, but that would not provide any concrete advantage in terms of control-flow analysis.

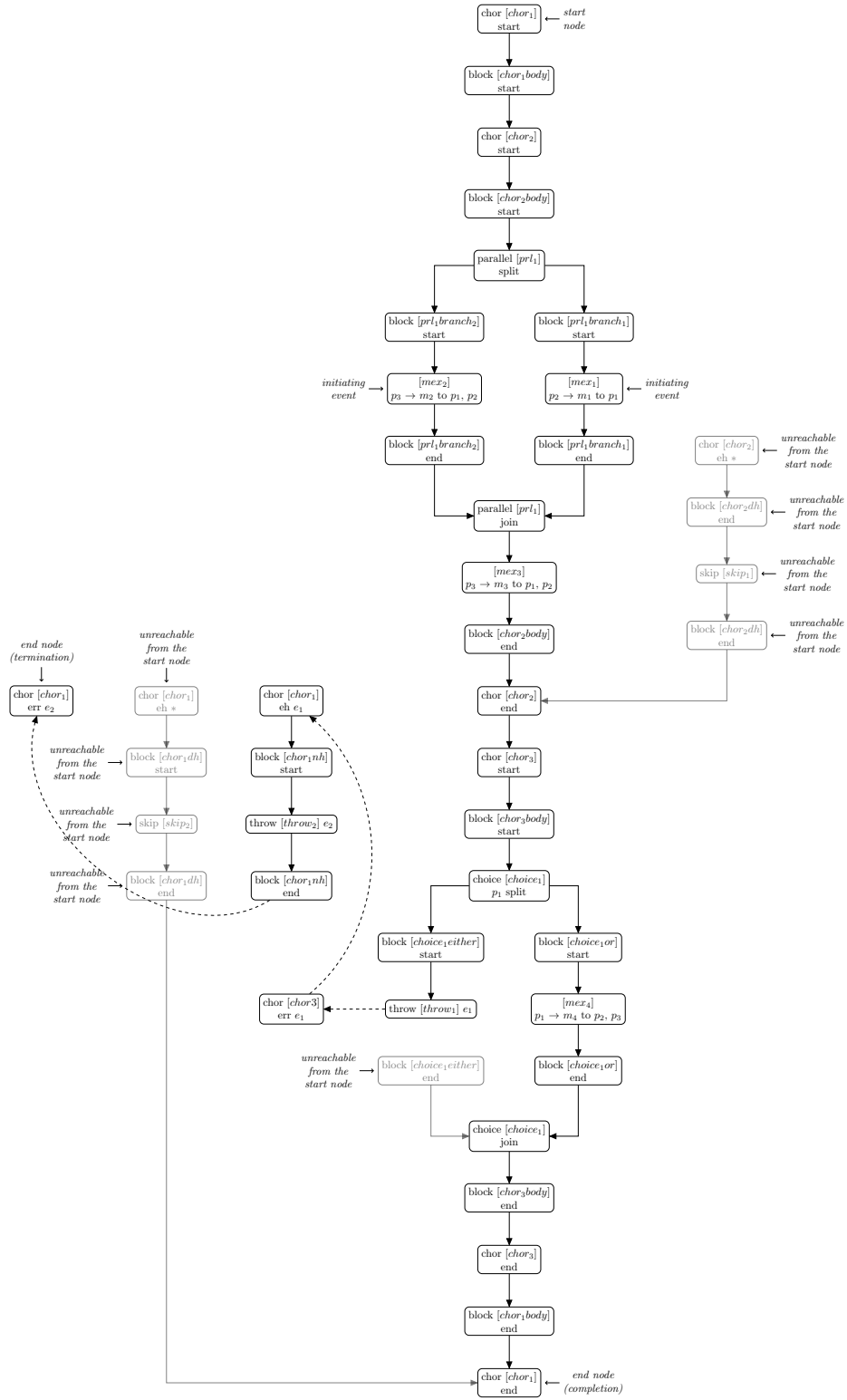


Figure 3.15: Control Flow Graph of the choreography shown in Figure 3.3.

3.4.2 Reconciling Actions and Events in Enactments

The aim of this section is twofold. First, it provides a formal interpretation of the ordering of events as specified by the edges in CFGs of ChorTex choreographies. Second, it relates the events represented by nodes in CFGs with the actions performed by participants that constitute an enactment trace (see Section 3.3). Figure 3.16 summarizes the terminology and relationships between the terms that are introduced in this section.

Definition 3.6 (Preceding and Succeeding Events). An event e , represented in the CFG by a node n_e , *precedes* an event e' , represented in the CFG by a node $n_{e'}$, if there is a control-flow edge with source n_e and target $n_{e'}$ in the CFG. If e precedes e' , e is said to be a *predecessor* of e' . Conversely, if e precedes e' , then e' *succeeds* e or, equivalently, e' is a *successor* of e .

The precedence of events expressed by the edges in CFGs is a partial order, but does not represent necessary causality. If an event e precedes e' in the CFG, then in an enactment the firing of e *might* cause the firing of e' . That is, the firing on an event does not necessarily cause the firing of its successors. For example, the event that represents the taking of the decision in a choice activity is predecessor to all the events that represent the beginning of the enactment of one of the choice's branches. However, every time a decision is taken, only one the start events of those branches may occur, and possibly *none* if the choice activity is terminated because of the propagation of an exception.

We distinguish between two types of events: *participant-activated* and *reactive*.

Definition 3.7 (Participant-Activated Events and Nodes). The firing of a *participant-activated* event represents the performing of an action by some of the participants. The types of participant-activated events are the following:

- Dispatching of a message;
- Enactment of an opaque activity by the participants involved in it;
- Decision of which branch of a choice activity to enact;
- Decision of whether to iterate the body of an iteration activity.

A node representing the firing of a participant-activated event is said to be a *participant-activated node*.

Each type of participant-activated events corresponds to one of the types of actions specified in Definition 3.1. Consistently with the fact that the reception of a message by a recipient is not considered an action (see Definition 3.1), the latter is not represented as an event either.

Definition 3.8 (Reactive Events and Nodes). *Reactive events* are fired as the result of the firing of participant-activated events. The types of reactive events are all those not listed as participant-activated in Definition 3.7. A node that represents the firing of a reactive event is said to be a *reactive node*.

To exemplify reactive and participant activate events, consider the CFG shown in Figure 3.17 obtained from a straightforward choreography.

The enactment of the choreography represented by the CFG in Figure 3.17 begins when the participant p dispatches the message m_1 to the participant p' . The dispatching of the message m_1 , and therefore the firing of the participant-activated event n_{m_1} , triggers the firing of the reactive events n_{c_1} , n_{b_1} and n_{b_2} (fired in this order). The events n_{c_1} and n_{b_1} are “retroactively” fired because the dispatching of m_1 is an initiating action (see Definition 3.4). Since the enactment does not exist before m_1 is dispatched, the firing of n_{c_1} and n_{b_1} is implied to “fill the gap” in the traversal of the CFG from the start node to n_{m_1} . After the firing of n_{b_2} , the enactment “waits” for the dispatching of m_2 by p' . The dispatching of the message m_2 triggers the reactive events n_{b_3} , n_{b_4} and n_{c_2} , the latter representing the end of the enactment.

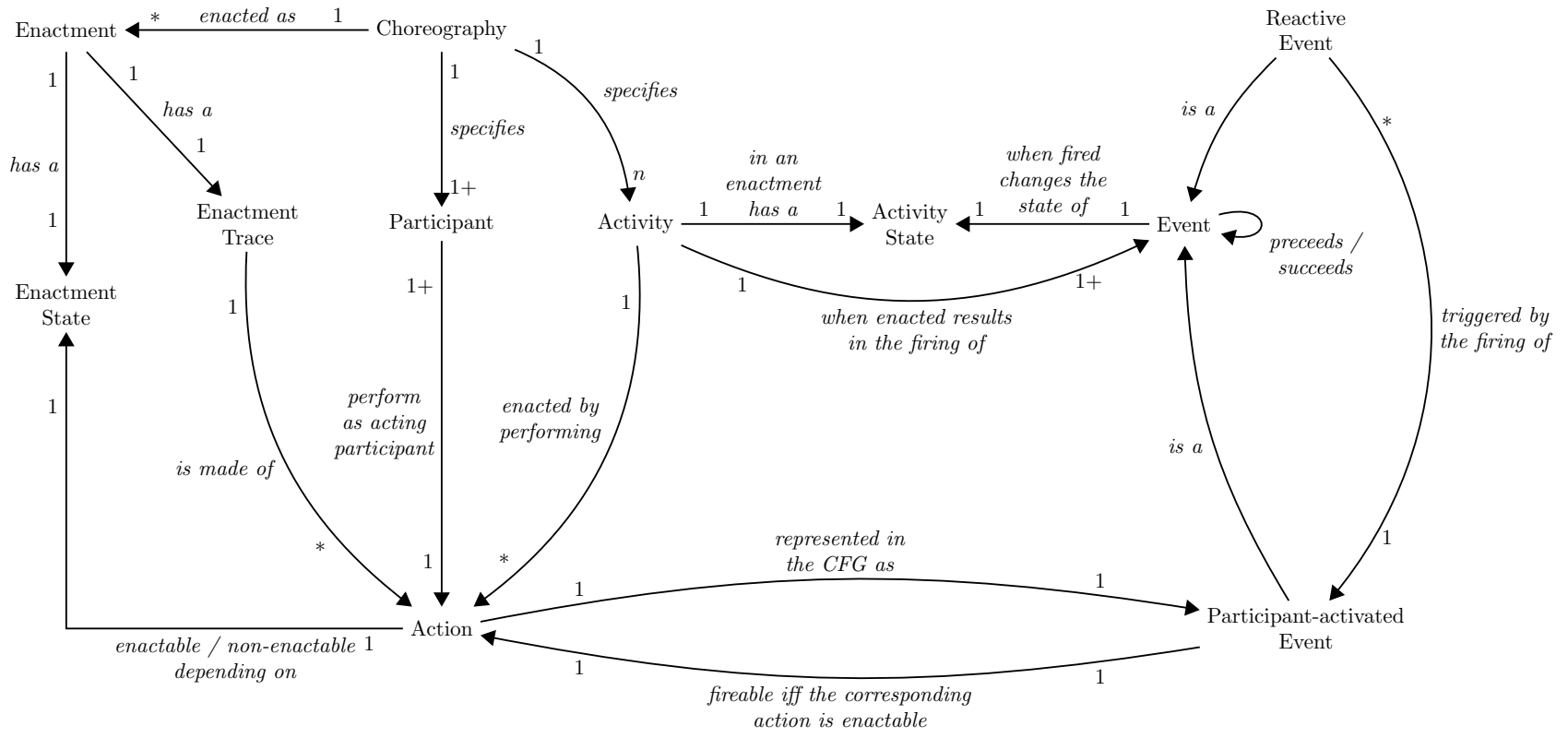


Figure 3.16: Terminology related to enactments, actions and events.

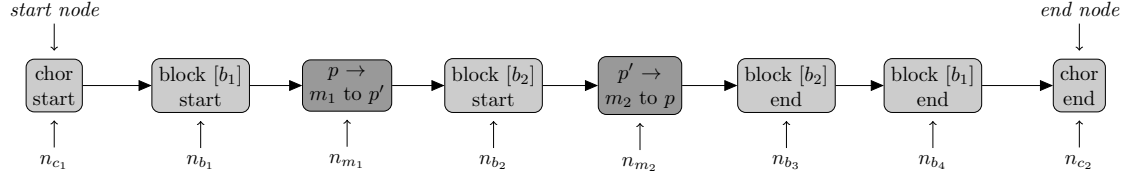


Figure 3.17: A simple CFG exemplifying reactive nodes, colored light grey, and participant-activated ones, colored in a darker shade of grey.

Definition 3.9 (Choreography Violations). An action \mathbf{a} performed by some participants in the enactment state χ is said to *violate* the choreography if there is no rule of the operational semantics of ChorTex that is applicable to χ and that results in \mathbf{a} to be appended to the enactment trace.

For example, considering the CFG in Figure 3.17, if p' were to send m_2 to p before receiving m_1 , that would violate of the choreography. In other words, a violation of a choreography consists in the performing an action in an enactment state that, according to how the choreography is specified and the operational semantics of ChorTex, is not allowed. In order to avoid violations of the choreography, acting participants have to abide restrictions in terms of *which enactment states* allow their actions can be performed.

Definition 3.10 (Enactable Actions). An action \mathbf{a} is *enactable* in the enactment state χ if and only if \mathbf{a} performed by its acting participants in χ does not violate the choreography.

Building on the notion of enactable activity, it can be specified when a participant-activated event is *fireable*.

Definition 3.11 (Fireable Events). The participant-activated event e_a associated with the action \mathbf{a} is *fireable* in the enactment state χ if \mathbf{a} is enactable in χ .

That is, a participant-activated event is fireable if its acting participants can trigger it without, so doing, violating the choreography.

3.5 Well-Formedness Requirements

This section provides a definition of well-formedness of ChorTex choreographies based on a number of well-formedness requirements. Due to limitations of the DSL used in Section 3.2 to specify the syntax of ChorTex presented in Section 3.2 specifies the structure of all possible *syntactically correct* ChorTex choreographies.

Well-formedness Requirement 1 (Syntactic Correctness). The ChorTex choreography does not violate any of the syntactical constraints specified by the grammar presented in Section 3.2.

Syntactical correctness is a pre-requisite for well-formedness. However, the syntax of ChorTex allows to specify choreographies that, while syntactically correct, nevertheless violate some of the additional requirements specified in the remainder of this section.

Well-formedness Requirement 2 (Unique Activity Names). All the activity names in a ChorTex choreography are unique, i.e., there are no two activities with the same name.

Figure 3.18 specifies a recursive function that extracts the names of the activities of a choreography. If the list resulting from applying the function to the root choreography contains no duplicates, the Well-formedness Requirement 2 is satisfied. Activity names are optional, therefore the rules in Figure 3.18 have two cases for each activity type, one with and one without names.

The ChorTex syntax allows choreographies in which multiple named handlers catch the same exception. While it would be an interesting feature of the language – multiple blocks triggered by

$names(\mathbf{skip})$	$:= \emptyset$
$names(\mathbf{skip} \ [skip])$	$:= [skip[$
$names(p \rightarrow m \ \mathbf{to} \ p_1, \dots, p_n)$	$:= \emptyset$
$names([mex] \ p \rightarrow m \ \mathbf{to} \ p_1, \dots, p_n)$	$:= [mex[$
$names(\mathbf{opaque} \ (p_1, \dots, p_n))$	$:= \emptyset$
$names(\mathbf{opaque} \ [o] \ (p_1, \dots, p_n))$	$:= [o[$
$names(\mathbf{throw} \ e)$	$:= \emptyset$
$names(\mathbf{throw} \ [t] \ e)$	$:= [t[$
$names(\{\mathbf{A}_1; \dots; \mathbf{A}_n\})$	$:= names(\mathbf{A}_1) \circ \dots \circ names(\mathbf{A}_n)$
$names([b] \ \{\mathbf{A}_1; \dots; \mathbf{A}_n\})$	$:= \{b\} \circ names(\mathbf{A}_1) \circ \dots \circ names(\mathbf{A}_n)$
$names(\mathbf{throw} \ [t] \ e)$	$:= [t[$
$names(\mathbf{choice} \ p \ \mathbf{either} \ \mathbf{A}_1 \ \mathbf{or} \ \dots \ \mathbf{or} \ \mathbf{A}_n)$	$:= names(\mathbf{A}_1) \circ \dots \circ names(\mathbf{A}_n)$
$names(\mathbf{choice} \ [c] \ p \ \mathbf{either} \ \mathbf{A}_1 \ \mathbf{or} \ \dots \ \mathbf{or} \ \mathbf{A}_n)$	$:= [c[\circ names(\mathbf{A}_1) \circ \dots \circ names(\mathbf{A}_n)$
$names(\mathbf{iteration} \ p \ \mathbf{do} \ \mathbf{A})$	$:= names(\mathbf{A})$
$names(\mathbf{iteration} \ [i] \ p \ \mathbf{do} \ \mathbf{A})$	$:= [i[\circ names(\mathbf{A})$
$names(\mathbf{parallel} \ \mathbf{do} \ \mathbf{A}_1 \ \mathbf{and} \ \dots \ \mathbf{and} \ \mathbf{A}_n)$	$:= names(\mathbf{A}_1) \circ \dots \circ names(\mathbf{A}_n)$
$names(\mathbf{parallel} \ [prl] \ \mathbf{do} \ \mathbf{A}_1 \ \mathbf{and} \ \dots \ \mathbf{and} \ \mathbf{A}_n)$	$:= [prl[\circ names(\mathbf{A}_1) \circ \dots \circ names(\mathbf{A}_n)$
$names(\mathbf{chor} \ (\mathbf{A}_b \ \ e_1 : \mathbf{A}_1 \ \ \dots \ \ e_n : \mathbf{A}_n \ \ * : \mathbf{A}_*))$	$:= names(\mathbf{A}_b) \circ names(\mathbf{A}_*) \circ names(\mathbf{A}_1) \circ \dots \circ names(\mathbf{A}_n)$
$names(\mathbf{chor} \ [chor] \ (\mathbf{A}_b \ \ e_1 : \mathbf{A}_1 \ \ \dots \ \ e_n : \mathbf{A}_n \ \ * : \mathbf{A}_*))$	$:= [chor[\circ names(\mathbf{A}_b) \circ names(\mathbf{A}_1) \circ \dots \circ names(\mathbf{A}_n) \circ names(\mathbf{A}_*)$

Figure 3.18: Definition of the $names(\mathbf{A})$ function which returns the list of activity names declared by the activity \mathbf{A} and the activities nested into it (if any).

$msgTypes(\text{skip})$	$:= \emptyset$
$msgTypes(p \rightarrow m \text{ to } p_1, \dots, p_n)$	$:= \{m\}$
$msgTypes(\text{opaque } (p_1, \dots, p_n))$	$:= \emptyset$
$msgTypes(\text{throw } e)$	$:= \emptyset$
$msgTypes(\{\mathbf{A}_1; \dots; \mathbf{A}_n\})$	$:= msgTypes(\mathbf{A}_1) \cup \dots \cup msgTypes(\mathbf{A}_n)$
$msgTypes(\text{choice } p \text{ either } \mathbf{A}_1 \text{ or } \dots \text{ or } \mathbf{A}_n)$	$:= msgTypes(\mathbf{A}_1) \cup \dots \cup msgTypes(\mathbf{A}_n)$
$msgTypes(\text{iteration } p \text{ do } \mathbf{A})$	$:= msgTypes(\mathbf{A})$
$msgTypes(\text{parallel do } \mathbf{A}_1 \text{ and } \dots \text{ and } \mathbf{A}_n)$	$:= msgTypes(\mathbf{A}_1) \cup \dots \cup msgTypes(\mathbf{A}_n)$
$msgTypes(\text{chor } (\mathbf{A}_b \mid e_1 : \mathbf{A}_1 \mid \dots \mid e_n : \mathbf{A}_n \mid * : \mathbf{A}_*))$	$:= msgTypes(\mathbf{A}_b) \cup msgTypes(\mathbf{A}_*) \cup \left(\bigcup_{i \in [1, n]} msgTypes(\mathbf{A}_i) \right)$

Figure 3.19: Definition of the $msgTypes(\mathbf{A})$ function which returns the set of message types declared by the activity \mathbf{A} and the activities nested into it (if any).

the same exception being thrown – it can be realized by nesting a parallel activity inside one single named exception handler. For simplicity in defining the remediation analysis method in Chapter 4 and the remediation strategies in Chapter 5, ChorTex should avoid multiple ways of expressing the same semantics; therefore, we introduce the following well-formedness requirement:

Well-formedness Requirement 3 (Unique Named Exception Handler Types). All named exception handlers for a ChorTex choreography catch different types of exceptions. Formally, the following logic predicate must hold for every choreography $\mathbf{chor}(\mathbf{A}_b \mid e_1 : \mathbf{A}_1 \mid \dots \mid e_n : \mathbf{A}_n \mid * : \mathbf{A}_*)$:

$$\forall i, j \in [1, n] : i \neq j \Rightarrow e_i \neq e_n$$

This predicate must hold for the root choreography as well to all the nested ones.

As discussed in Section 3.1, to satisfy the “Unequivocal message exchanges” design assumption we need to specify the following well-formedness requirement:

Well-formedness Requirement 4 (Unique Message Types). No two message exchanges in a choreography declare the same message type.

The message types declared in a choreography are straightforwardly extracted by the activity \mathbf{A} using the function $msgTypes(\mathbf{A})$ defined in Figure 3.19 that is specified in a very similar way as the one specified in Figure 3.18 to retrieve the names of the activities. Notice that this requirement could be relaxed, for example by requiring that message exchanges that declare the same message type have disjoint sets of receivers. That is, every participant can receive messages of one time in the scope of at most one message exchange. However, this would increase the complexity of the realizability analysis method presented in Chapter 4, which would have to deal with realizability defects stemming from potential misinterpretation by the message recipients of which message exchange activity has been enacted and, therefore, fallacious assumptions on the current state of the enactment. This type of realizability defect is known in the state of the art as “non-deterministic choice” (see Section 2.3.4.4). It simply does not seem worth the trouble to face this type of issues in ChorTex given the fact that, as long as the message type identifier is included as meta-data in the message itself, two different message types can specify equivalent data-structures and this requirement would still be satisfied.

Other two corner-cases that are not covered by the syntax of ChorTex concern message exchanges. First, the syntax allows one participant to be both sender and recipient of one message exchange. This possibility is not appropriate for an interaction modeling language such as ChorTex: a message sent by a participant to itself is more an *internal action* than a proper message exchange, and internal actions are not in the scope of interaction choreographies (see Section 2.1.3.2).

Well-formedness Requirement 5 (Sender is not a Recipient). No participant is both sender and recipient of any one message exchange. Formally, the following predicate must hold for every message exchange $p_s \rightarrow m \mathbf{to} p_{r_1}, \dots, p_{r_n} : p_s \notin \{p_{r_1}, \dots, p_{r_n}\}$

The second case we need to rule out consists of one participant appearing multiple times in the list of recipients for a message exchange activity.

Well-formedness Requirement 6 (Distinct Recipients in Message Exchange Activities). The participant identifiers appearing as recipient in any message exchange activity must all be all distinct. Formally, given the message exchange activity $p \rightarrow m \mathbf{to} p_1, \dots, p_n$, the following must hold:

$$\forall i, j \in [1, n] : i \neq j \Rightarrow p_i \neq p_j$$

This well-formedness requirement is mostly motivated by usability for the end user. If the ChorTex allowed to specify the same participant identifier multiple times in a message exchange captivity, the end user might assume that the message is delivered multiple times to that participant (which is not the case, see the local projection of participants specified later in Section 4.5). Alternatively, allowing “redundant” participant identifiers in the recipient list of message exchanges

would almost certainly lead to modeling errors. In particular, the modeler might simply not notice to have entered the same participant identifier multiple times (for example, due to a “cut & paste” oversight). Spotting such an issue would be far from straightforward, in particular when the participant identifiers used in a choreography are not very different from each other or meaningful in the scope of the model.¹⁴

For the same reasons that motivate the “Distinct Recipients in Message Exchange Activities” requirement, distinct participant identifiers are necessary in opaque activities:

Well-formedness Requirement 7 (Distinct Participants in Opaque Activities). The participant identifiers appearing as participants in any opaque activity must all be distinct. Formally, given the message exchange activity **opaque** (p_1, \dots, p_n) , the following must hold:

$$\forall i, j \in [1, n] : i \neq j \Rightarrow p_i \neq p_j$$

The wellformedness requirements specified so far could all have been incorporated in the ChorTex grammar specified in (Section 3.2). The following wellformedness requirement, instead, is related with the operational semantics of the exception handling mechanism of ChorTex, and require the concepts related to events and actions previously introduced in Section 3.4.2.

Well-formedness Requirement 8 (Univocal Exception Triggering). From every participant-activated node in the CFG of a choreography must be reachable at most one **throw** e reactive node without traversing another participant-activated event.

In a nutshell, this requirement can be phrased in natural language as follows: any one action performed by participants in a choreography may trigger the throwing of at most one exception. Consider the choreography snippet and the corresponding CFG shown in Figure 3.20. When enacting this choreography snippet, the dispatching of the message m by p_s to p_r would trigger concurrently the throwing of two exceptions of type e_1 and e_2 , respectively. Similarly to the case of multiple named exception handlers in one choreography catching the same exception type, we rule out this corner-case to simplify the remainder of this work.

Based on the wellformedness requirements defined so far, we finally provide the following definition of wellformedness of ChorTex choreography:

Definition 3.12 (Well-formedness of ChorTex choreographies). A ChorTex choreography is well-formed if and only if it satisfies all of the following well-formedness requirements:

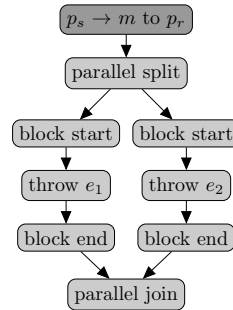
- Syntactic Correctness

```

1 ...
2  $p_s \rightarrow m$  to  $p_r$ ;
3 parallel do {
4   throw  $e_1$ 
5 } and {
6   throw  $e_2$ ;
7 }
8 ...

```

(a) Choreography snippet in which one action triggers the throwing of two exceptions.



(b) Control Flow Graph of the choreography snippet.

Figure 3.20: Example of multiple exceptions triggered by one action.

¹⁴This example should, by all rights, cause cold shivers down the spine of those readers with a background of software development. In particular, those readers who, at one point or another, have been “blessed” with fellow developers that did not believe in “descriptive variable names.”

- Unique Activity Names
- Unique Named Exception Handler Types
- Unique Message Types
- Sender is not a Recipient
- Distinct Recipients in Message Exchange Activities
- Distinct Participants in Opaque Activities
- Univocal Exception Triggering

In the perspective of the design of choreography modeling languages, it is worth mentioning two well-formedness requirements that were initially considered, but did not “make it to the list.” Both these requirements had to do with unreachable nodes in the Control Flow Graphs, namely:

- No activities can be specified in a block after a throw activity, i.e., there can at most one throw activity directly nested in a block, and it must be the last nested activity;
- All exception handlers must be reachable.

Both these requirements are aimed at preventing “dead code” in ChorTex choreographies, i.e., activities that would never be enacted in a choreography. On the one hand, the rationale for avoiding “dead activities” is clear: the smaller the choreography, the more readable it is and, therefore, the less likely it is for the modeler to perform modeling mistakes. On the other hand, are usually a stepping stone in the modeling of a choreography. For example, one may want to specify first the logic that handles exceptions, and then the one that throws it. Alternatively, the exception throwing activity may simply represent a placeholder for choreography logic not yet written.¹⁵ It might seem counterintuitive, or perhaps too reminiscent of general-purpose programming languages, but the freedom to “misuse” exception handling in such ways seems to actually be beneficial in the scope of modeling complex choreographies. If we were to add these two wellformedness requirements to Definition 3.12, this would prevent implementations of the realizability analysis method presented in Chapter 4 from running realizability analyses and, through it, supporting the modeler in creating realizable choreographies. Which, quite frankly, is a rather bad idea when, by means of the CFGs and reachability analysis, it is very easy (and implemented in the prototype, see Chapter 6) to simply display *warnings* about the dead activities to the modeler.

3.6 Differences between ChorTex and Chor

At first sight, ChorTex adopts a natural language-like syntax, while its predecessor Chor [205] has the “look & feel” of a formalism. This author believes ChorTex’s syntax to be more user-friendly than Chor’s. (However, there is no scientific evidence to support this claim.)

The differences between Chor and ChorTex, however, go deeper than just the syntax, and are treated in the remainder of this section.

3.6.1 Opaque versus Internal Activities

Chor provides a construct called *basic activity* which allows to specify *internal activities* executed by single participants. Chor’s basic activities are not to be confused with ChorTex’s: in ChorTex, the term “basic activity” is used to collectively denote activities that do not allow others to be nested in them, i.e., skip, message exchange, opaque and throw activities, see Figure 3.2. The execution of Chor’s basic activities is instantaneous, infallible (i.e., it never results in an exception being thrown),

¹⁵A bit like the throwing of `UnsupportedOperationException` that peppers Java code automatically-generated using many frameworks.

and, since it does not generate message exchanges, is not registered in the enactment trace. The rationale of Chor’s basic activities in choreographies is not discussed in [205]; presumably, basic activities are included in Chor because of its “twin” orchestration language Role. In fact, during the process of projection, a basic activity specified in the choreography is transformed in an internal activity the peer of the participant that executes it.

Allowing the specification of internal activities in a choreography language focusing on interaction modeling seems to be a rather questionable design decision. Interaction modeling of choreographies focuses on the *global* behavior of the choreography, detailing the *public* actions of the participants. Instead, Chor’s basic activities are internal – that is, *private* – to the participants that execute them. Unlike the choice and iteration constructs, whose internal activities (the decisions) do affect the enactment, basic activities do not influence the sequencing of activities in the choreography, and are therefore semantically void. Therefore, in ChorTex we substitute Chor’s basic activities with the opaque ones, which drop the assumption of instantaneous execution and involve multiple participants.

3.6.2 Choreography Nesting versus Referencing

In Chor, choreographies are uniquely identified by name and can be referenced from inside other choreographies using the `perf` construct. When the `perf Cm` activity is executed, the choreography identified by C_m is enacted. The completion of C_m leads to the completion of `perf Cm`. Similarly, the termination of C_m because of the propagation of an uncaught exception results in that exception propagating outside `perf Cm`. In ChorTex choreography referencing is replaced by choreography nesting. Choreography referencing simplifies the modular definition of choreographies, which is a useful feature for a choreography modeling language. However, it also complicates considerably the exposition of the realizability analysis method presented in Section 4 by requiring the adoption of inter-procedural flow analysis techniques instead of basic flow analysis ones. Our decision is taken for reasons of understandability of the exposition. Nevertheless, it is possible to extend the realizability analysis to support choreography referencing in ChorTex by means of inter-procedural flow analysis techniques, e.g., [159, 57, 170, 158].

3.6.3 Finalization Handlers

An extension to Chor presented in [205] allows the specification of finalization behaviors, i.e., activities that are executed irrespective of the completion of the body of a choreography or its termination due to exception propagation. Similarly to the case of choreography referencing, ChorTex does not include such functionality for reasons of understandability of the exposition. Known techniques of CFG analysis such as [189] can deal with finalization handling and should be an easy task to adapt them for supporting finalization handlers in ChorTex choreographies.

Chapter 4

Awareness-Based Realizability Analysis of ChorTex Choreographies

This chapter presents a realizability analysis method for ChorTex choreographies. As previously discussed in Section 2.3, the foundation of any realizability analysis method is the realizability definition it aims at verifying. In this work, we adapt to ChorTex a classical definition of realizability called *strong realizability*, which is presented in Section 4.1.

Figure 4.1 provides an overview of the realizability analysis method for strong realizability of ChorTex choreographies that is presented throughout this chapter. First is generated from the choreography a Control Flow Graph (CFG) that represents the events that occur in enactments (e.g., the beginning or completion of an activity) and their ordering using the production rules that have been introduced in Section 3.4. In Section 4.3, the CFG is transformed into an Awareness Model (AWM) by annotating its nodes with information on the *participant awareness*. Participant awareness, introduced in Section 4.2, is a symbolic representation of which events can each participant observe. Using Control Flow Graphs as model for the ordering of events enables the calculation of participant awareness using very performant techniques borrowed from the field of control-flow analysis for programming languages. The strong realizability of the choreography is verifying using *awareness constraints*, i.e., constraints on the participant awareness that are specified in Section 4.4.

Each type of awareness constraint identifies a type of realizability defects that can occur in ChorTex choreographies, which are catalogued in Section 4.5. In Section 4.5 we also investigate the relationship between realizability defect types, participant awareness and constructs of choreography modeling languages.

Section 4.6 revisits some of the design decisions of ChorTex (presented in Section 3.1) in the

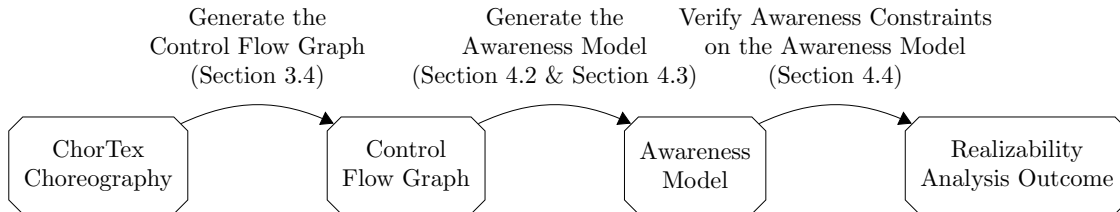


Figure 4.1: An outlook of the realizability analysis of ChorTex choreographies; the chamfered rectangles represent artifacts, and the arrows connecting them are steps of the realizability analysis.

light of the realizability analysis method presented throughout this chapter.

This chapter is concluded by Section 4.7, which discusses the computational complexity of our realizability analysis method and, based on the constraints defined in Section 4.4, builds a case against using interrupting constructs like exception throwing in choreographies.

4.1 Strong Realizability of ChorTex Choreographies

The definition of choreography realizability adopted in this work is based on the following notion of *conversation*.

Definition 4.1 (Conversations). A *conversation* is an enactment trace (see Definition 3.2) restricted to only the actions that describe interactions among the participants, i.e., message exchanges and opaque activities (which *may* involve interactions among the participants).

$$\text{conv}(\langle a \rangle \circ \sigma) := \begin{cases} \langle a \rangle \circ \text{conv}(\sigma) & \text{if } a = p \xrightarrow{m} p_1, \dots, p_n \\ \langle a \rangle \circ \text{conv}(\sigma) & \text{if } a = \text{III}(p_1, \dots, p_n) \\ \text{conv}(\sigma) & \text{otherwise} \end{cases}$$

$$\text{conv}(\langle \rangle) := \langle \rangle$$

The function $\text{conv}(\sigma)$ defined above specifies how to extract a conversation from an enactment trace σ . The symbol $\langle \rangle$ denotes the empty enactment trace.

Put simply, a conversation is an enactment trace “purged” of all internal actions, i.e., decisions taken by decision makers in the scope of choice and iteration activities. Besides the definition of conversation, we must first introduce an important assumption about *how the participants behave* in the scope of choreography enactments.

Definition 4.2 (Well-behaved Participant). A participant p is well-behaved if it satisfies in all choreography enactments the following two conditions:

No willful violation If the participant p knows that the enactment state is s and that performing the action a in the enactment state s would violate the choreography, p will not perform a in the enactment state s ;

No harmful inaction If p knows that without its performing of the action a the enactment cannot reach any final state, p will *eventually* perform a .

In other words, well-behaved participants meet all the obligations specified by their roles, which is not an unexpected hypothesis given the fact that a choreography enactment is a highly collaborative endeavor. At the best of our knowledge, virtually every work in the state of the art of choreography realizability assumes well-behaved participants (though not always the assumption is explicitly stated), with the only exception being [33]. It should be noted that the “no willful violation” and “no harmful inaction” are fundamentally aspects of the safety (“nothing bad happens”) and liveness (“something good eventually happens”) properties that are ubiquitous in the literature on concurrent programming languages and model checking. More specifically, the “no willful violation” and “no harmful inaction” assumptions could be specified in terms of safety and liveness applied to the relationship between the role specified by the choreography and the peer implementation that plays that role.

The following definition of realizability that is adopted throughout this thesis is the adaptation to ChorTex choreographies of the definition of *strong realizability* proposed in [116]:

Definition 4.3 (Strong realizability of ChorTex choreographies). A ChorTex choreography is *strongly realizable* if it is well-formed, stuck-free and exist peers that, if well-behaved, once composed result in a composition that is language equivalent in terms of conversations to the choreography.

According to the styles of realizability definitions that have been explored in Section 2.3.1.1, our definition of strong realizability is mixed/existential. Figure 4.2 characterizes it using the dimensions of realizability definitions presented in Section 2.3.1.2.

As its name implies, strong realizability is a rather strict definition of choreography realizability. It requires that it is possible to create participant implementations that can enact all and only the conversations specified by the choreography. This definition is consistent with the usages of choreographies in the practice of SOA and BPM, in particular with the fact that choreographies are often used as technical contracts among the participants (i.e., the “Specification and communication” choreography usage discussed in Section 2.1.2).

4.2 Participant Awareness in Choreographies

As discussed in Section 3.3, a ChorTex choreography is fundamentally the intensional specification of enactment traces, that is, sequences of actions such as message exchanges and internal decisions that the participants are tasked to perform during enactments. The Control Flow Graph of a ChorTex choreography represents the ordering of the firing of events representing, for example, the beginning and completion of an activity. Events are fired during an enactment as a result of the actions performed by the participants (see Section 3.4.2).

When enacting a choreography, the participants are subject to the phenomenon of *blindness* [88, 93] (also known as “myopia” [188]), i.e., they might be able to observe only some of the actions that are performed by other participants. Because of blindness, it might be the case that some participants have insufficient information on how the enactment proceeds in order to perform their roles as mandated by the choreography. For example, it may happen that during one enactment a participant *does not know* that its performing of a certain action is necessary for the progress of the enactment; since the action is not taken, the enactment becomes stuck. Put in terms of events, a participant might be unable to observe sufficient events to play its role as specified, which is symptomatic of the lack of realizability of the choreography that is being enacted.

The goal of this section is to provide symbolic means for describing *which events a participant can and cannot observe* during the enactments of a ChorTex choreography. This information is the basis for the verification of the strong realizability presented in Section 4.4.

<p>Choreography Modeling Language: ChorTex</p> <p>Communication Model: Asynchronous (see Design Assumption 1):</p> <ul style="list-style-type: none"> Sending strategy: non-blocking sending, always succeed; Buffer: The participant implementations of ChorTex choreographies use queues with the following characteristics: <ul style="list-style-type: none"> Ordering: ordered Capacity: unbounded Quantity: multiple buffers, one per participant; buffers are specific to single enactments (messages of the same type sent during different enactments are queued up in different buffers) <p>Behavioral Similarity Relation: Language equivalence of conversations</p> <p>Peer Composition Method: Cartesian product</p> <p>Intrinsic Properties: Well-formedness, stuckness freedom</p>

Figure 4.2: Characterization of strong realizability of ChorTex choreographies in terms of dimensions of realizability definitions.

Definition 4.4 (Observable Actions). The participant p can observe the performing of an action \mathbf{a} (equivalently, \mathbf{a} is *observable* by p), if and only if:

- p is an acting participant of \mathbf{a} (see Definition 3.1), or
- \mathbf{a} is a message exchange and p is one of its recipients.

In any other case, the participant p cannot observe the performing of the action \mathbf{a} .

In other words, a participant is able to observe only those actions it itself performs as acting participant, such as the dispatching of a message, or that have observable consequences for the participant, namely the reception of a message addressed to it. Therefore, the participants have *local projections* of enactment traces.

Definition 4.5 (Local Projections). The *local projection* of the participant p of an enactment trace σ is the trace obtained by purging from σ the actions not observable by p using the function $\pi(p, \sigma)$ as defined in Figure 4.3.

It should be noted that the local projection of a participant on an enactment trace does not necessarily match the order in which the participant has observed the actions. Consider, for example, the running example shown in Figure 3.3: due to the delay between the dispatching of a message and its processing by the receivers, it might be that the participant p_1 processes the message m_2 before m_3 , even though they have been dispatched by the respective senders in the opposite order.

Due to their local projection, participants have a different knowledge on which actions have been performed up to that point in an enactment. Therefore, participants will have different knowledge of which events have been fired so far and may be fired in the future. The knowledge of a participant with respect of the firing of an event is captured by the concept of *participant awareness*. We identify the two following types of participant awareness with respect to an event e :

Participant awareness by p of e becoming fireable represents the capability of p of observing that e becomes fireable, i.e., that the firing of e in the current enactment state would not violate the choreography (see Definition 3.11).

Participant awareness by p of the firing of e represents the capability of p of observing that e has been fired.

The differentiation of the participant awareness between the “fire-ability” of the event e and its actual firing enables us to describe situations in which a participant p knows that e may be fired in the current enactment state, therefore causing the transition to another enactment state, even when the actual firing of e is not observable by p . This is extremely important for enacting choreographies without violations. For example, it may be the case that the action \mathbf{a} is enactable in the current enactment state χ , but not in the enactment state χ' that results from the firing of an event e which was fireable in χ . The participants rely solely on their local projections to decide when to perform the actions that they *believe* are enactable. Therefore p may unknowingly violate the choreography by performing \mathbf{a} in the enactment state χ' because it did not know that the enactment state had transitioned from χ to χ' due to the firing of e .

Table 4.1 presents the four possible *participant awareness states*, i.e. the “extents” of participant awareness that a participant may have about an event e the becoming fireable and being fired.

Immediate awareness, denoted by “ia,” means that the participant knows *as soon as it happens* that the event becomes fireable or is fired. For example, the sender of a message exchange is immediately aware that the message has been dispatched. Similarly, the decision maker of a certain decision is immediately aware when that decision has been taken.

Eventual awareness, denoted by “ea,” is a “relaxed” form of immediate awareness. The participant does not necessarily know that the event has become fireable or has been fired the moment it happens, but will be able to observe it “sooner or later,” i.e., *eventually* as understood in Linear

$$\pi(p, (\mathbf{a}) \circ \sigma) := \begin{cases} (\mathbf{a}) \circ \pi(p, \sigma) & \text{if } (\mathbf{a} = [mex]p_s \xrightarrow{m} p_{r_1}, \dots, p_{r_n}) \wedge (p = p_s \vee p \in \{p_{r_1}, \dots, p_{r_n}\}) \\ (\mathbf{a}) \circ \pi(p, \sigma) & \text{if } (\mathbf{a} = [o] \parallel (p_1, \dots, p_n)) \wedge p \in \{p_1, \dots, p_n\} \\ (\mathbf{a}) \circ \pi(p, \sigma) & \text{if } (\mathbf{a} = [c] p \xrightarrow{?} \mathbf{x}) \\ (\mathbf{a}) \circ \pi(p, \sigma) & \text{if } (\mathbf{a} = [i] p \xrightarrow{\circ} \mathbf{x}) \\ \pi(p, \sigma) & \text{otherwise} \end{cases}$$

$$\pi(p, ()) := ()$$

Figure 4.3: The definition of the $\pi(p, \sigma)$ function that extracts the local projection of the participant p from the enactment trace σ .

Awareness state	Notation	Applied to	Description
Immediately aware	$\frac{p}{ia}$	e becoming fireable	p can observe that e is fireable as soon as e becomes fireable
		firing of e	p can observe the firing of e as soon as e is fired
Eventually aware	$\frac{p}{ea}$	e becoming fireable	p is able to observe that e becomes fireable immediately when or at some point after e has become fireable
		firing of e	p is able to observe the firing of e immediately when or at some point after e has been fired
Unaware	$\frac{p}{ua}$	e becoming fireable	p cannot observe that e becomes fireable
		firing of e	p cannot observe the firing of e
Not involved	$\frac{p}{ni}$	e becoming fireable	p is not yet involved in the enactment when e becomes fireable
		firing of e	p is not yet involved in the enactment when e is fired

Table 4.1: The possible participant awareness states of the participant p with respect to the event e .

Temporal Logics (LTL) [21]. For example, due to the communication model assumed by ChorTex, the recipient of a message will eventually know (at the moment of the reception) that the message has been dispatched. Straightforwardly, immediate awareness implies eventual awareness.

Unawareness, denoted by “ua,” means that the unaware participant cannot observe at all an event becoming fireable or being fired. For example, a participant that is neither sender nor recipient of a message is unaware of the event that represents the dispatching of that message. Of course, due to some later interaction with the other participants, a participant might be able to “imply” that a certain message has been dispatched even though it could not observe the dispatching event. This is the case, for example, when the dispatching of the message m must have necessarily been occurred before another message m' is dispatched, and p can observe the dispatching of m' but not the one of m . However, this type of implication is not of interest to the realizability analysis method proposed in this work and therefore has no effect on the definition of the participant awareness states.

Finally, a participant is *not involved*, denoted by “ni,” with respect an event becoming fireable or been fired if, when that happens, the participant may have not yet taken part in the enactment. A participant takes part in an enactment the moment it performs its first action or it receives a message in the scope of that enactment. Non-involvement implies unawareness: if a participant has not yet partaken an enactment, it cannot be aware of any of the enactment events that have taken place thus far.

4.3 Annotating Participant Awareness States

This section presents how to create Awareness Models (AWMs) by annotating Control Flow Graphs generated from ChorTex choreographies (see Section 3.4) with the participant awareness states introduced in the previous section. Since the AWM is simply a CFG with additional annotations, the nodes and control-flow edges of an AWM remain exactly the same as those of the original CFG. The annotations on the participant awareness states associated with the nodes in the AWMs are called *node-awareness states*.

Definition 4.6 (Node-Awareness States). In the Awareness Model of a choreography that specifies the participants p_1, \dots, p_m , the *node-awareness state* of a node n_e that represents the firing of the event e is defined as the following couple:

$$\left(AW_{\text{fireable}}(n_e), AW_{\text{fired}}(n_e) \right)$$

$AW_{\text{fireable}}(n_e)$ and $AW_{\text{fired}}(n_e)$ are sets comprising m items, one per participant, each representing the participant awareness state of one participant with respect to the event e becoming fireable and e being fired, respectively. Assuming a choreography that defines m participants, the $AW_{\text{fireable}}(n)$ and $AW_{\text{fired}}(n)$ of a node n are both formally defined as follows:

$$\left\{ \frac{p_i}{\alpha} : \forall i \in [1, m] \alpha \in \{\text{ia}, \text{ea}, \text{ua}, \text{ni}\} \right\}$$

That is, $AW_{\text{fireable}}(n)$ and $AW_{\text{fired}}(n)$ contain *exactly one* participant awareness state associated to each of the participants specified by the choreography. For example, given the node n_e of a choreography that specifies the participants p_1 and p_2 , the following node-awareness state means that p_1 is immediately aware of the event e becoming fireable and e being fired, while p_2 is unaware of e becoming fireable but eventually aware of e having been fired:

$$AW(n_e) := \left(\left\{ \frac{p_1}{\text{ia}}, \frac{p_2}{\text{ia}} \right\}, \left\{ \frac{p_1}{\text{ua}}, \frac{p_2}{\text{ea}} \right\} \right)$$

In the remainder, $AW_{\text{fireable}}(n, p)$ and $AW_{\text{fired}}(n, p)$ denote the participant awareness state of the participant p in $AW_{\text{fireable}}(n)$ and $AW_{\text{fired}}(n)$, respectively.

4.3.1 The Awareness Annotation Algorithm

The function $\mathbb{P}(chor)$, defined in Figure 4.4, returns the set of participant identifiers appearing in the choreography $chor$ and is recursively defined on the the syntax of ChorTex. Figure 4.5 presents the pseudo-code of the Awareness Annotation Algorithm that calculates the node-awareness states associated with the nodes of an Awareness Model. The Awareness Annotation Algorithm (AAA) has the classic structure of fixed-point algorithms on Control Flow Graphs. It begins with an initialization phase that assigns for each node and to all participants an initial “non-involved” AW_{fireable} value. For each participant p , the AW_{fired} of a node n is calculated by the function $f(n, p)$, which is discussed in detail in Section 4.3.3. In a nutshell, the function $f(n, p)$ calculates how the firing of the event represented by the AWM node n affects the participant awareness state of the participant p .

After the initialization phase, the algorithm iterates until a fixed point is reached, i.e., until further iterations produce no further modifications of the node-awareness states. With each iteration step, a node n is chosen at random to be processed. (Although, refined implementations of the Awareness Annotation Algorithm may employ heuristics to select which node to process based, for example, on the cycles in the Control Flow Graphs; however, such heuristics fall outside the scope of this work, as they add nothing to the concepts here presented.) The processing of the node n consists of the following two steps, which are repeated for each participant p specified in the choreography. First, the AW_{fireable} value of n is recalculated as the *safe approximation* of all the AW_{fired} values of n ’s predecessors. This safe approximation is calculated using the function \boxplus described in Section 4.3.2. $AW_{\text{fired}}(n, p)$ is then updated with the outcome of $f(n, p)$, which is calculated on the basis of the updated value of $AW_{\text{fireable}}(n, p)$ resulting from the previous step. It should also be noticed that not all nodes are processed to reach the fixed point. The nodes that are not reachable from the start node through a path of control flows are ignored when calculating the node-awareness states because they represent events that are never fired in any enactment of the choreography. In the pseudo-code of the Awareness Annotation Algorithm, this is realized by the “**reachable from start**” clauses on Line 9 and Line 21.

Figure 4.6 presents the Awareness Model resulting from applying the Awareness Annotation Algorithm to the Control Flow Graph of the running example shown in Figure 3.3. The node-awareness states are represented as labels attached to the respective nodes. Notice that there are no AW_{fireable} and AW_{fired} values annotated on the nodes that are not reachable from the start node in any possible enactment, which are drawn less markedly in Figure 3.3, as those nodes are ignored by the Awareness Annotation Algorithm.

$$\mathbb{P}(\mathbf{A}) := \begin{cases} \emptyset & \text{if } \mathbf{A} = \text{skip} \\ \{p_s, p_{r_1}, \dots, p_{r_n}\} & \text{if } \mathbf{A} = p_s \rightarrow m \text{ to } p_s, p_{r_1}, \dots, p_{r_n} \\ \{p_1, \dots, p_n\} & \text{if } \mathbf{A} = \text{opaque } (p_1, \dots, p_n) \\ \emptyset & \text{if } \mathbf{A} = \text{throw } e \\ \bigcup_{i \in [1, n]} \mathbb{P}(\mathbf{A}_i) & \text{if } \mathbf{A} = \{\mathbf{A}_1, \dots, \mathbf{A}_n\} \\ \{p\} \cup \left(\bigcup_{i \in [1, n]} \mathbb{P}(\mathbf{A}_i) \right) & \text{if } \mathbf{A} = \text{choice } p \text{ either } \mathbf{A}_1 \text{ or } \dots \text{ or } \mathbf{A}_n \\ \{p\} \cup \mathbb{P}(\mathbf{A}) & \text{if } \mathbf{A} = \text{iteration } p \text{ do } \mathbf{A}' \\ \bigcup_{i \in [1, n]} \mathbb{P}(\mathbf{A}_i) & \text{if } \mathbf{A} = \text{parallel do } \mathbf{A}_1 \text{ and } \dots \text{ and } \mathbf{A}_n \\ \mathbb{P}(\mathbf{A}) \cup \left(\bigcup_{i \in [1, n]} \mathbb{P}(\mathbf{A}^{e_i}) \right) \cup \mathbb{P}(\mathbf{A}^*) & \text{if } \mathbf{A} = \text{chor } (\mathbf{A}' \mid e_1 : \mathbf{A}^{e_1} \mid \dots \mid e_n : \mathbf{A}^{e_n} \mid * : \mathbf{A}^*) \end{cases}$$

Figure 4.4: Recursive definition of the function $\mathbb{P}(\mathbf{A})$ for extracting the participants involved in an activity \mathbf{A} .

Awareness Annotation Algorithm

```

1 function generateAwarenessModel (choreography chor)
2   returns awareness model {
3     /* generate chor's Control Flow Graph as described in Section 3.4 */
4     awareness model awm := generateCFG(chor);
5     /* retrieve the start node of the Awareness Model */
6     node start := startNode(awm);
7
8     /* initialization */
9     foreach node n in  $\mathbb{N}(\text{awm})$  reachable from start do
10      /*  $\mathbb{P}(\text{chor})$  returns the set of participants */
11      /* specified by the choreography chor (see Figure 4.4) */
12      foreach p in  $\mathbb{P}(\text{chor})$  do
13         $\text{AW}_{\text{fireable}}(n, p) := \text{ni}$ ;
14         $\text{AW}_{\text{fired}}(n, p) := f(n, p)$ ;
15      end foreach
16    end foreach
17
18    /* iteration until a fixed-point is reached */
19    repeat
20      /*  $\mathbb{N}(\text{awm})$  returns the set of nodes in the awm awareness model */
21      foreach node n in  $\mathbb{N}(\text{awm})$  reachable from start do
22        /* update  $\text{AW}_{\text{fireable}}$  for each participant as safe approximation */
23        /* of the  $\text{AW}_{\text{fired}}$  of the predecessors of n (see Section 4.3.2) */
24        /*  $\mathbb{P}(\text{chor})$  returns the set of participants defined in chor */
25        foreach p in  $\mathbb{P}(\text{chor})$  do
26           $\text{AW}_{\text{fireable}}(n, p) := \biguplus_{n' \in \text{pred}(n)} (\text{AW}_{\text{fired}}(n', p))$ ;
27          /* Outcome of  $f(n, p)$  depends on  $\text{AW}_{\text{fireable}}(n, p)$  */
28           $\text{AW}_{\text{fired}}(n, p) := f(n, p)$ ;
29        end foreach
30      end foreach
31    until no  $\text{AW}_{\text{fired}}(n, p)$  changes
32
33    return awm;
34  }

```

Figure 4.5: Pseudo-code of the Awareness Annotation Algorithm.

4.3.2 Safe Approximation of $\text{AW}_{\text{fireable}}$ Values of Nodes

In the iteration phase of the Awareness Annotation Algorithm, the $\text{AW}_{\text{fireable}}$ value of a node is recalculated for each participant as the safe approximation of the AW_{fired} values for the same participant of the predecessors of the node. The safe approximation of AW_{fired} of a set of nodes is calculated by the function \biguplus (read as “meet”) defined Figure 4.7.¹ The safe approximation is calculated according to the following principles:

1. Immediate awareness implies eventual awareness (see Section 4.2);
2. Non involvement implies unawareness (see Section 4.2);

¹The meet function is traditionally denoted in flow-analysis algorithms by the symbol \bigwedge . However, we need the \bigwedge symbol later in this work to represent the logic conjunction of multiple predicates; hence our unconventional choice of symbol for denoting the meet function.

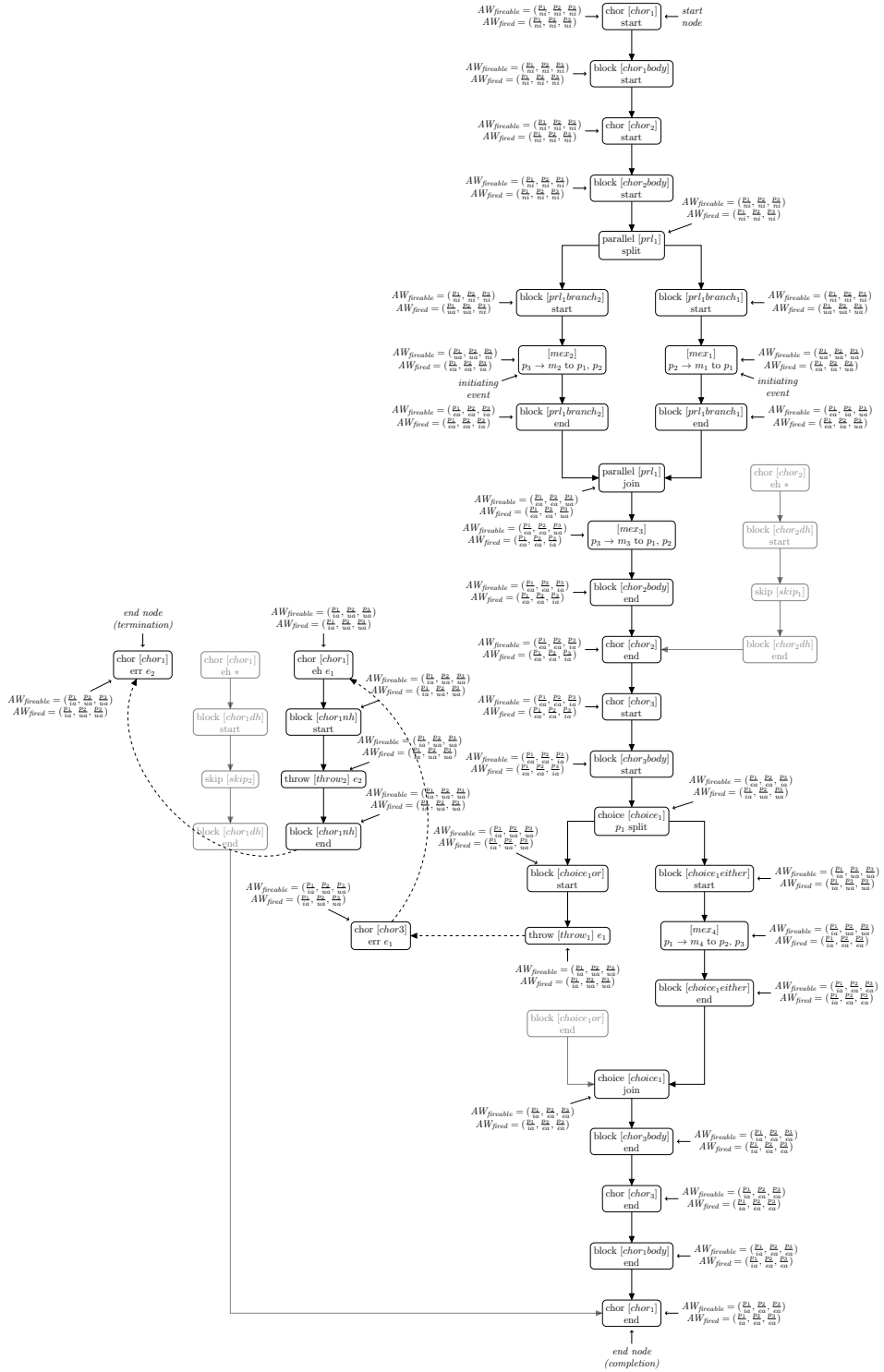


Figure 4.6: Awareness Model obtained by applying the Awareness Annotation Algorithm to the Control Flow Graph shown in Figure 3.15.

$$\begin{aligned} & \alpha \in \{\text{ia}, \text{ea}, \text{ua}, \text{ni}\} \\ \widehat{\text{A}}(\alpha_1, \dots, \alpha_m) := & \begin{cases} \text{ia} & \text{if } \forall j \in [1, m] : \alpha_j = \text{ia} \\ \text{ea} & \text{if } (\exists j \in [1, m] : \alpha_j = \text{ea}) \wedge \\ & (\forall i \in [1, m] : \alpha_j = \text{ia} \vee \alpha_j = \text{ea}) \\ \text{ni} & \text{if } \forall j \in [1, m] : \alpha_j = \text{ni} \\ \text{ua} & \text{otherwise} \end{cases} \end{aligned}$$

Figure 4.7: Definition of the $\widehat{\text{A}}$ function that calculates the safe approximation of the AW_{fired} values of multiple nodes.

3. When a participant is marked as unaware in any of the input awareness states, it is also marked as unaware in their safe approximation.

The last principle comes from the “nature” of the information aggregated by $\widehat{\text{A}}$, namely participant awareness states. The goal is to safely approximate the participant awareness state of one participant with respect to a certain event becoming fireable. As discussed in Section 3.4.2, an event becomes fireable when one or more of its predecessors have been fired (see Definition 3.11). If the participant p is unaware of the firing any of the predecessors of the event e represented in the AWM by the node n , then there are some enactments of the choreography in which p is unaware of e becoming fireable. Therefore, the safe approximation consists of marking p as unaware in $\text{AW}_{\text{fireable}}(n)$.

4.3.3 Updating the AW_{fired} Value of Nodes

The value of $\text{AW}_{\text{fired}}(n, p)$ of a node n for the participant p is calculated by the function $f(n, p)$, which is defined in Table 4.2 in a case-based fashion depending on the different types of Awareness Model nodes. In a nutshell, the function $f(n, p)$ specifies how the firing of the event represented by the node n affects the participant awareness state of the participant p , and it is applied in the Awareness Annotation Algorithm on each node n once for each of the participants in the choreography. In some cases, $f(n, p)$ simply returns the value of $\text{AW}_{\text{fireable}}(n, p)$, meaning that the traversal of the node n does not modify the participant awareness state of the participant p . Specifically, this is the case of nodes representing the enactment of skip and throw activities, end of block activities, merge and split nodes of parallel activities, merge nodes of choice activities as well as the start, end, exception handling and error propagation nodes of choreographies. Intuitively, since none of these nodes represent actions, there is no possible modification to the participant awareness due to blindness or due to participants observing their own actions.

After the traversing of a node representing the dispatching of a message exchanges, the sender becomes immediately aware. The recipients, instead, are eventually aware due to the asynchronous communication model assumed by ChorTex choreographies (see Section 3.1). The participants that were not involved before the message exchange and that do not take part into it, remain not involved after it. In all other cases, the participants remain unaware or become unaware due to blindness.

The traversing of nodes that represent opaque activities makes all the participants that partake them eventually aware. This comes from the assumption that the participants partaking an opaque activity reckon its completion (see Section 3.2). Participants that were not involved before the opaque activity and that do not participant in it, stay not involved. Otherwise, the participants that were involved before the opaque activity, but that do not take part in it, become unaware.

AWM node n	Definition of $f(n, p)$
$p_s \rightarrow m$ to p_{r_1}, \dots, p_{r_n}	$\begin{cases} \text{ia} & \text{if } p = p_s \\ \text{ea} & \text{if } p \in \{p_{r_1}, \dots, p_{r_n}\} \\ \text{ni} & \text{if } (p \notin \{p_s\} \cup \{p_{r_1}, \dots, p_{r_n}\}) \wedge \text{AW}_{\text{fireable}}(n, p) = \text{ni} \\ \text{ua} & \text{otherwise} \end{cases}$
opaque (p_1, \dots, p_m)	$\begin{cases} \text{ea} & \text{if } p \in \{p_1, \dots, p_m\} \\ \text{ni} & \text{if } p \notin \{p_1, \dots, p_m\} \wedge \text{AW}_{\text{fireable}}(n, p) = \text{ni} \\ \text{ua} & \text{otherwise} \end{cases}$
block start	$\begin{cases} \text{ua} & \text{if block is branch of parallel activity } \mathbf{A} \wedge \text{AW}_{\text{fireable}}(n, p) = \text{ni} \wedge \\ & p \text{ is acting participant in a reachable node of another branch of } \mathbf{A} \\ \text{AW}_{\text{fireable}}(n, p) & \text{otherwise} \end{cases}$
choice p_i split	$\begin{cases} \text{ia} & \text{if } p = p_i \\ \text{ni} & \text{if } p \neq p_i \wedge \text{AW}_{\text{fireable}}(n, p) = \text{ni} \\ \text{ua} & \text{otherwise} \end{cases}$
iteration p_i	$\begin{cases} \text{ia} & \text{if } p = p_i \\ \text{ni} & \text{if } p \neq p_i \wedge \text{AW}_{\text{fireable}}(n, p) = \text{ni} \\ \text{ua} & \text{otherwise} \end{cases}$
otherwise	$\text{AW}_{\text{fireable}}(n, p)$

Table 4.2: The definition of $f(n, p)$ for the node n and the participant p , case-based on the various types of AWM nodes.

The nodes representing local decisions for iteration and choice activities modify the participant awareness in a way similar to opaque activities. Namely, since the decision is local, the decision maker is immediately aware of it. All participants that do not take the local decision remain not involved, if they were so before the local decision is taken; otherwise, they become unaware.

Finally, the start nodes of blocks that are branches of parallel activities require special care in the case of participants that are not involved in the AW_{fireable} . Branches of a parallel activity are enacted concurrently and, due to the flow algorithm used to annotate the participant awareness, the branches cannot “synchronize” on the participants becoming involved. That is, if the participant p is not involved before the beginning of a parallel activity, but it becomes involved during the enactment of one of its branches, the flow algorithm would propagate the information about p ’s non-involvement on all other branches. Therefore, if p is an acting participant in some action of a branch, marking it as non involved in another branch would violate the “safe approximation” principle of participant awareness, because non-involvement is *more specific* than unawareness. In the definition of $f(n, p)$ shown in Table 4.2, this case is dealt with by marking a participant p as unaware of the start node n of a block that is a branch of a parallel activity if p is not involved in the $AW_{\text{fireable}}(n)$, and p is an acting participant in a reachable node in any other branch of the parallel activity. (Non-reachable nodes with acting participants naturally do not count, because they are never executed.) This is the case, for example, of the AWM node `block [prl1branch2] start` in the running example shown in Figure 4.6. In fact, the participants p_1 and p_2 are unaware in $AW_{\text{fired}}(\text{block [prl₁branch₁] start})$ – instead of non-involved as reported in $AW_{\text{fireable}}(\text{block [prl₁branch₂] start})$ – because p_1 and p_2 are acting participants in some other branch of the parallel activity prl_1 , namely the block activity $prl_1branch_1$.

4.3.4 Computational Complexity of the AAA

The Awareness Annotation Algorithm is a “classic” iterative, fixed-point algorithm. Similar fixed-point algorithms, like the ones for calculating dominance and post-dominance on Control Flow Graphs, perform a number of iterations to update the values associated to the nodes proven in [193, 74] to be between $\mathcal{O}(|E(\text{awm})| \times \log |E(\text{awm})|)$ and $\mathcal{O}(|N(\text{awm})|^2)$, where $E(\text{awm})$ and $N(\text{awm})$ denote the sets of control-flow edges and nodes of the AWM awm , respectively. In the reminder we assume the latter, higher complexity, as an approximation of how many iterations are performed by the AAA to reach the fixed-point. Additionally, the AAA requires the execution of a reachability analysis between every couple of nodes, which is known to have upper-bound complexity $\mathcal{O}(\log^2 |N(\text{awm})|)$.

Each invocation of the merge function $\hat{\text{A}}$ has complexity linear with the number participants and the number of successors of the node whose input-value is being calculated. AWMs are generally structured so that each nodes has few successors/predecessors (see Section 3.4), so we can approximate the upper bound complexity of $\hat{\text{A}}$ to $\mathcal{O}(|P(\text{chor})|)$, where $|P(\text{chor})|$ denotes the cardinality of the set of participants, that is, how many participants are altogether declared in the choreography.

The definition of the update-function $f(n, p)$ is case-based. The evaluation of all its cases can be considered to be $\mathcal{O}(1)$, with the exception of the case in which n is a `block start` node generated from a branch of a parallel activity; in this last case, the evaluation requires the traversal of the parse-tree of the other branches, and it can be conservatively estimated to $\mathcal{O}(|A(\text{chor}) - 4|)$. The reason for this unusual term is simple: the activities that can be specified in the other branches of the parallel can be at most $|A(\text{chor}) - 4|$, because for sure they do not contain the current branch, the one activity that the branch necessarily contains (see ChorTex’s syntax BNF in Section 3.2), the parallel activity in which the current branch is nested and the root choreography. In practice, the amount of activities in other branches tends to be much smaller than that.

Putting all the pieces together, the AAA has an upper-bound complexity of:

$$\mathcal{O}(|N(\text{awm})|^2 \times |P(\text{chor})| \times |A(\text{chor}) - 4| + \log^2 |N(\text{awm})|)$$

Since we are calculating the upper-bound complexity, we can simplify the estimation above as

follows:

$$\mathcal{O}(|\mathbb{N}(awm)|^2 \times |\mathbb{P}(chor)| \times |\mathbb{A}(chor)|)$$

Surely, an over-approximation of 4 on the count of activities and a complexity of $\log^2 |\mathbb{N}(awm)|$ do not matter much, given the fact that the dominant term is $|\mathbb{N}(awm)|^2$; put formally:

$$|\mathbb{N}(awm)|^2 \gg \log^2 |\mathbb{N}(awm)|$$

(The sign \gg reads “much bigger than.”) Finally, we can observe that the number of nodes in a AWM is roughly linear with the number of activities specified by the choreography. (It is not possible to quantify precisely the ratio between activities and AWM nodes, as it depends on which type of activities are specified by the choreography.) In fact, basic and iteration activities are mapped to sub-graphs with only one node. Choreographies, instead, are mapped to sub-graphs with at least two nodes (start and end) plus one node for each declared exception handler and propagating exception. The other activities are mapped to two nodes. In other words, the amount of nodes and activities are comparable, but there are strictly more nodes than activities (this is guaranteed by the root choreography alone, which results, at the very least, in the start and end node for itself, its body, plus one node for a nested message exchange activity). Therefore, the estimation of the upper-bound complexity of the AAA can be further conservatively simplified to:

$$\mathcal{O}(|\mathbb{N}(awm)|^3 \times |\mathbb{P}(chor)|)$$

Finally, an important consideration that affects fixed-point algorithms like the Awareness Annotation Algorithm is whether they converge, i.e., if it is always possible to reach a fixed-point in a finite amount of steps. In a nutshell, the Awareness Annotation Algorithm always reaches a fixed-point because the \mathbb{A} and $f(n, p)$ functions are such that the participant-awareness for a given participant with respect to a given node is monotonically non-decreasing over the iterations of the Awareness Annotation Algorithm according to the following strict order among participant-awareness states:

$$ni < ua < ea < ia$$

That is, if in one iteration of the Awareness Annotation Algorithm the participant-awareness of a participant for a node is, say, ea , then it is not possible for it to become ua or ni in other iterations. Given an iteration over all the nodes of the AWM awm , there are two cases: no participant-awareness value is modified, and thus the fixed-point has been reached, or some participant-awareness state has changed. Since the changes to participant-awareness states are monotonically non-decreasing, the maximum possible amount of iteration steps to reach a fixed point is:

$$|\mathbb{N}(awm)| \times |\mathbb{P}(awm)| \times 2 \times 3$$

The first two factors are due to the assumption that, in the very worst case, each iteration results in exactly one modification to the participant-awareness state of exactly one participant (the bare minimum to prevent us from having reached a fixed point). The factor 2 comes from the fact that, for each node n and participant p , there are an $\text{AW}_{\text{fireable}}(n, p)$ and an $\text{AW}_{\text{fired}}(n, p)$ participant-awareness states. The factor 3 comes from the fact that there can be tops three transitions in the strict order among participant-awareness values.

4.4 Awareness Constraints for Verifying Strong Realizability

This section introduces constraints on participant awareness states of an AWM of a ChorTex choreography that are necessary and sufficient for ensuring its strong realizability. We assume the participants to be well-behaved (see Definition 4.2). Under the assumption of well-behaved participants, the strong realizability of a ChorTex choreography is verified if the two following *realizability requirements* are met for every action specified by the choreography:

Realizability Requirement 1 (Observe your actions becoming enactable). The acting participants of every action are immediately or eventually aware of that action becoming enactable.

Realizability Requirement 2 (Observe your actions becoming not enactable). The acting participants of every action are immediately aware of the latter becoming not enactable.

The two requirements above are of straightforward explanation and can be collectively summarized as “know when you can enact your actions.” Firstly, if a well-behaved participant can observe when its actions become enactable and when they become non-enactable, that participant will never violate the choreography. Secondly, if the performing of an action is necessary for the progress of the enactment, its acting participants know that the action can (and should) be performed and, since they are well-behaved, they will eventually perform that action. If the “Observe your actions becoming enactable” and “Observe your actions becoming not enactable” requirements are met by a well-formed choreography, that choreography is strongly realizable (the proof is provided in Chapter A).

In order to verify the strong realizability of a ChorTex choreography, the two requirements are “translated” into *awareness constraints*.

Definition 4.7 (Awareness Constraints). An *awareness constraint* is a logic predicate evaluated on the node-awareness states of an Awareness Model.

Besides the usual predicate logic operators, an awareness constraint admits the following functions that “extract” the participant awareness state of a participant p with respect to the event e represented by the node n becoming fireable and being fired:

$$AW_{\text{fireable}}(n, p) = \alpha \quad (\text{Participant awareness state in } AW_{\text{fireable}})$$

$$AW_{\text{fired}}(n, p) = \alpha \quad (\text{Participant awareness state in } AW_{\text{fired}})$$

As in Section 4.3, the symbol α denotes one of the four participant awareness states described in Section 4.2, namely:

$$\alpha \in \{\text{ia, ea, ua, ni}\}$$

Each awareness constraint is *attached* to one particular AWM node. This allows us to specify the following “short-hand” notations that make implicit which AWM node is being considered:

Long hand	Short hand
$AW_{\text{fireable}}(n, p) = \alpha$	$AW_{\text{fireable}}(p) = \alpha$
$AW_{\text{fired}}(n, p) = \alpha$	$AW_{\text{fired}}(p) = \alpha$

In particular, the couples of predicates shown to the right are equivalent in an awareness constraint attached to the AWM node n .

The remainder of this section is organized as follows. The awareness constraints for verifying the “Observe your actions becoming enactable” realizability requirement are presented in Section 4.4.1 and those for “Observe your actions becoming not enactable” in Section 4.4.2. Figure 4.8 exemplifies on the running example the types of awareness constraints that are presented in Section 4.4.1 and Section 4.4.2.

4.4.1 Observe Your Actions Becoming Enactable

The “Observe your actions becoming enactable” realizability requirement is translated to awareness constraints attached to nodes that represent participant-activated events. There are two types of such awareness constraints, depending on whether the action associated with the participant-activated event is initiating or not (see Definition 3.4). We treat first the case of non-initiating actions because this simplifies the overall exposition.

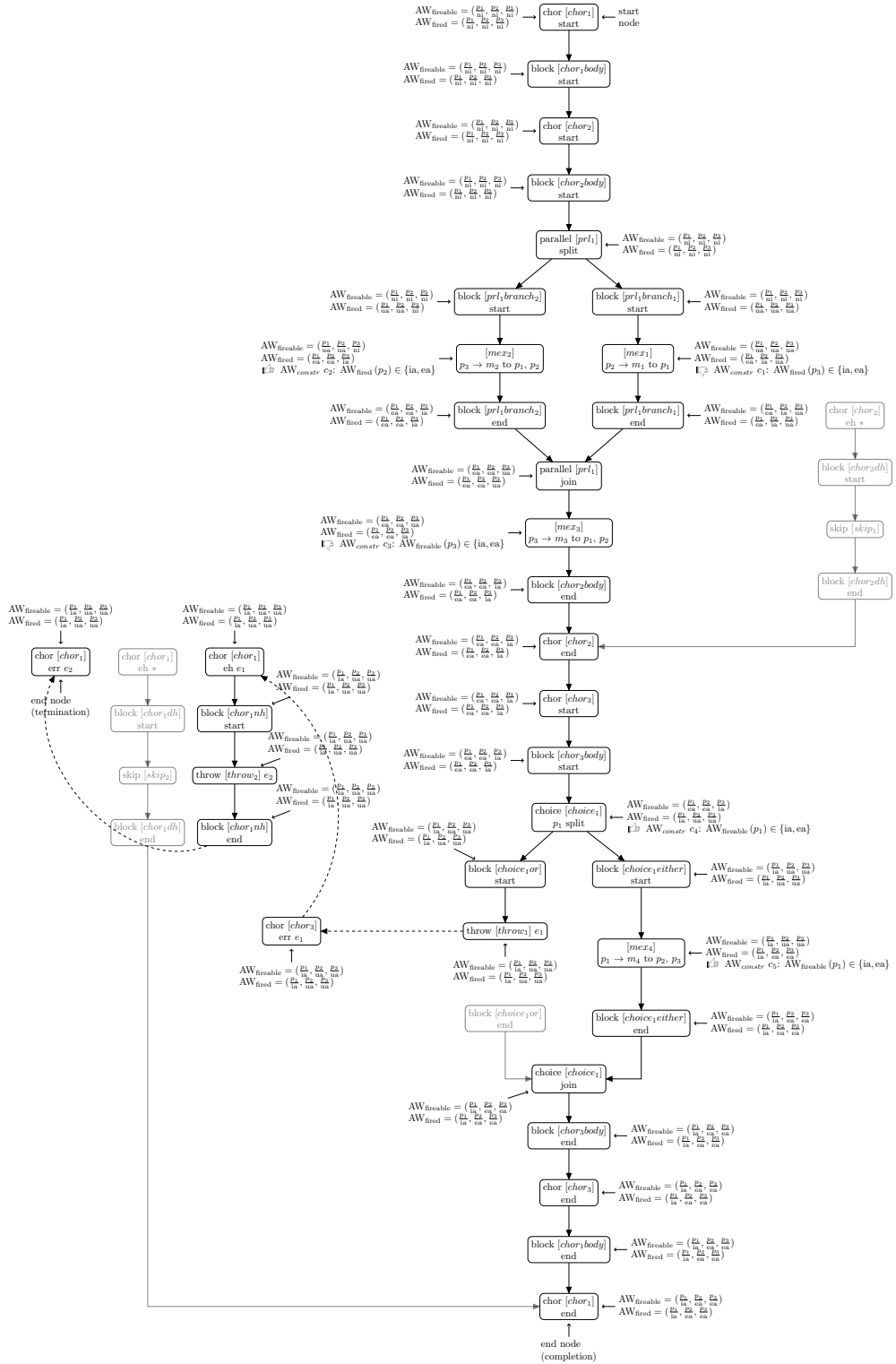


Figure 4.8: The Awareness Model presented in Figure 3.3; the satisfied awareness constraints are marked with a 👍 symbol, the unsatisfied ones in with a 👎.

4.4.1.1 Non-initiating actions

Let n be an AWM node representing the firing of a participant-activated event associated with the execution of the non-initiating action \mathbf{a} . To verify that the acting participants of \mathbf{a} can observe \mathbf{a} becoming enactable, n has attached, for each of its acting participants, one awareness constraint specified as follows:

$$p \in \text{actingParticipants}(\mathbf{a}) : \text{AW}_{\text{fireable}}(p) \in \{\text{ia}, \text{ea}\}$$

In a nutshell, these awareness constraints require that each acting participant of \mathbf{a} is immediately or eventually aware of \mathbf{a} becoming enactable. The function $\text{actingParticipants}(\mathbf{a})$ returns the acting participants of the action \mathbf{a} (see Definition 3.1). The rationale of these awareness constraints is the following. The acting participants must be aware of when they can perform the action, that is, when that action becomes enactable (hence the constraint on $\text{AW}_{\text{fireable}}$). If the acting participants are immediately aware of the action become enactable, they will be able to perform the action as soon as doing so would not violate the choreography. If they are eventually aware, they will be able to perform it at some point after the action becomes enactable. If the acting participants are unaware or not-involved, they can never perform that action for the risk of violating the choreography: this inability constitutes a realizability defect.

An example of this type of awareness constraint is c_3 in Figure 4.8. The participant p_3 is unaware of the fact that it can dispatch m_3 , and therefore it never will do it “for fear” of violating the choreography. This stalemate results in the enactment deadlocking. On the contrary, consider the awareness constraint c_4 in Figure 4.8. The decision maker p_1 of the choice activity choice_1 is eventually aware of the fact that it is required to decide which branch to enact. Assuming that it will perform the decision in a finite amount of time, this guarantees that no enactment will deadlock on this decision.

4.4.1.2 Initiating actions

Due to the constructs offered by the modeling language, ChorTex choreographies have either:

- A single initiating action;
- Multiple, *non mutually-exclusive* initiating actions.

Mutually-exclusive initiating actions, i.e., initiating actions exactly one of which occurs in every possible enactment, cannot be modeled with ChorTex. In fact, the only way to specify in ChorTex mutually exclusive actions is within branches of a choice activity. However, the decision action of the choice activity *dominates* (i.e., is always performed before) the actions in all the choice’s branches. This implies that an action in a branch of a choice activity can never be initiating. Therefore, ChorTex choreography can never have mutually-exclusive initiating actions.

In the case of a single initiating action, no awareness constraints are required to verify that its acting participants “observe their action becoming enactable.” The acting participants of the one initiating action perform create the enactment by performing that action. In a sense, the initiating action *is* the first source of participant awareness in an enactment. To perform it is reasonable not to require prior awareness, because there is no knowledge to be had of what has happened before in an enactment that *did not yet exist*. It should be noted that it is outside of the scope of realizability analysis method to determine how the acting participants of the initiating action decide to perform it. In other words, the conditions that lead to the initiation of an enactment are outside the scope of this realizability analysis method. This decision is particularly relevant for initiating opaque activities, which have multiple active participants. In this case, the participants would have to coordinate the beginning of the execution of the initiating opaque activity by means external to the choreography (e.g., a timer).

In the case of multiple, non-exclusive initiating actions, awareness constraints are required to ensure that all the initiating actions can be performed *regardless of which one is the first to occur in an enactment*. In the running example (see Figure 4.8) there are two initiating actions, namely

the dispatching of m_1 by p_2 to p_1 , and the dispatching of m_2 by p_3 to p_1 and p_2 . Either of these two actions may initiate an enactment of the choreography. Moreover, both actions *must* be performed in every enactment, because the parallel activity prl_1 does not complete until both messages have been dispatched. In each enactment, either m_1 is first dispatched and then m_2 is, or vice-versa. The dispatching of either message must provide “enough awareness” to sender of the other to be able to dispatch its message with the certainty of not violating the choreography. This is not the case in the running example. The dispatching of m_2 by p_3 to p_1 and p_2 makes the sender of m_1 , namely p_2 , eventually aware, and therefore able to later dispatch m_1 . On the other hand, consider the case in which the enactment is initiated by the dispatching of m_1 . The participant p_1 and p_2 are eventually and immediately aware of the dispatching of a message of type m_1 , respectively. However, since p_3 is not a recipient of m_1 , it does not know that the enactment has been initiated, and therefore it will not know that it must send m_2 . This causes a deadlock: since the message exchange activity mex_2 cannot be enacted (its acting participant is not aware that it can, and should, dispatch the message), the parallel activity prl_1 can never be completed, and the enactment is stuck.

Generalizing this previous example, the participant awareness resulting by the performing of each initiating action must enable the acting participants of all the other initiating actions to perform them. Put formally, given the set of initiating nodes n_{a_1}, \dots, n_{a_m} , each node n among them is associated with one or more of the following constraints:

$$p \in \left(\bigcup_{n' \in \{n_{a_1}, \dots, n_{a_m}\} \setminus \{n\}} \text{actingParticipants}(n') \right) : \text{AW}_{\text{fired}}(p) \in \{\text{ia}, \text{ea}\}$$

That is, for each of the acting participants *of the other initiating nodes*, there is one awareness constraint on n that requires that participant to be eventually or immediately aware of the traversing of n (and, therefore, of the performing of the action associated with it). If all awareness constraints of this type are satisfied, no matter which initiating action is the first to be performed in the enactment, all the others can be performed as well by their respective acting participants.

4.4.2 Observe Your Actions Becoming Not Enactable

When required to perform an action, such as generating a message or taking a decision, a participant is not required to take that action *immediately*. In the lifecycles of ChorTex activities presented in Section 3.3.2, this is captured by the PENDING states of message-exchange (Figure 3.5), choice (Figure 3.9) and iteration (Figure 3.10) activities. That is, some time may pass between the moment an action becomes enactable (see Definition 3.10) and when it is actually enacted. The awareness constraints that realize the “observe your actions becoming enactable” realizability requirement (presented in Section 4.4.1) ensure that the acting participants know when their actions become enactable. However, those awareness constraints do not account for the cases in which an action that is enactable at one point in time becomes later non-enactable due to changes to the enactment state. Specifically, actions change from enactable to non-enactable due to the throwing and propagation of exceptions (see Section 3.3.2). The operational semantics of exception throwing specifies that, as soon as an exception is thrown, all the other activities that are being enacted within the scope of the choreography catching that exception are immediately terminated. In such situations, the insufficient awareness of acting participants with respect to the ensued non-enactability of their actions may mislead them into violating the choreography. For example, consider the choreography shown in Figure 4.9. After the dispatching by participant p_1 to p_2 of the message of type m_1 , the parallel prl_1 activity is enacted, which in turn causes the enactment of the message exchange activity mex_2 and the choice activity c_1 . There are currently two enactable actions:

- The dispatching of a message of type m_2 by p_2 to p_1 ;
- The decision by p_1 on which branch of the choice activity c_1 to enact.

In an enactment in which p_1 decides for “true” in the choice activity c_1 before p_2 dispatches the message m_2 , the latter action becomes non-enactable. In this case, the choreography is violated if

p_2 dispatches a message of type m_2 to p_1 despite the fact that the exception e_1 has been thrown. In other words, the action representing the dispatch of m_2 is *interruptible* by the throwing of the exception of type e_1 .

Definition 4.8 (Interruptible actions and nodes). The action \mathbf{a} and its respective participant-activated node $n_{\mathbf{a}}$ are said to be interruptible by the throwing of the exception of type e , represented by the node n_e , if the traversal of n_e causes \mathbf{a} to change its status from enactable to non-enactable.

Identifying which participant-activated nodes are interruptible by a given **throw** e node is the key for specifying awareness constraints that ensure that participants do not violate the choreography by enacting actions that are non-enactable in the current enactment state.

4.4.2.1 Identifying Interruptible Nodes

To verify that a choreography does not have realizability defects such as that of the choreography in Figure 4.9, we need awareness constraints that verify that acting participants of interruptible nodes are immediately aware of the throwing or propagation of exceptions that would interrupt their actions. Figure 4.10 presents an algorithm that calculates such awareness constraints by exploiting the block-based structure of ChorTex to identify which parallel activities are terminated when a certain *propagates exception* event is fired in an enactment. The function $\text{parent}(\mathbf{A})$ returns the activity in which \mathbf{A} is nested. If \mathbf{A} is the root choreography (which, by definition, has no parent activity), $\text{parent}(\mathbf{A})$ returns NULL. The function $\text{sourceActivity}(n)$, instead, returns the activity from which the node n has been generated (see Section 3.4).

The algorithm in Figure 4.10 is divided in two phases. First, it identifies the parallel activity encompassing all those activities that may be terminated when the *propagates exception* node n is traversed. This parallel activity, called *outer parallel*, is the last one encountered navigating “upwards” (from nested to parent activities) the hierarchy of parents of n until the first choreography is found. (If there are parallel activities nested into each other without choreographies to “insulate” them, they are all terminated by the throwing of an exception in any of their branches.) It could be the case that no outer parallel is found. This is the case if:

- the activity that generates the *propagates exception* node n is not nested into any parallel activity (in which case the traversing of n makes no action non-enactable, since none may be enacted concurrently);

```

1  chor [chor1] (
2    {
3      [mex1] p1 → m1 to p2;
4      parallel [prt1] do {
5        [mex2] p2 → m2 to p1
6      } and {
7        choice [c1] p1
8        either {
9          throw [t1] e1
10         } or {
11          skip [s1]
12         }
13       };
14     [mex3] p2 → m3 to p1
15   }
16 )

```

Figure 4.9: A choreography in which the action of dispatching the message mex_2 can change from enactable to non-enactable due to the throwing of the of an exception of type e_1 .

```

1 function constraints (node n) returns list of constraints {
2   /* A is the activity that generates n */
3   activity A := sourceActivity(n);
4   /* identification of the “outer” parallel activity */
5   parallel activity outerParallel := NULL;
6   /* blacklist of other choice branches */
7   set of block activities blacklist :=  $\emptyset$ ;
8   activity A' := parent(A);
9   /* A' is NULL if A is the root choreography */
10  /* the iteration terminates when A' is a choreography that handles e */
11  while (A' not NULL &&
12    (A' is not choreography || A' not handles e)) do
13    if (A' is parallel do A1 and ... and An) then
14      outerParallel := A';
15    else if (A' is block &&
16      parent(A) = choice p either b1 or ... or bn) then
17      blacklist := blacklist  $\cup$  {b1, ..., bn} \ {A'};
18    end if
19    A' := parent(A');
20  end while
21  list of constraints res := [ ];
22  if outerParallel not NULL then
23    /* acting participants of concurrently enacted actions */
24    set of participants P :=  $\emptyset$ ;
25    /* retrieve chor's CFG sub-graph as described in Section 3.4 */
26    set of nodes N :=  $\mathbb{N}$ (generateCFG(outerParallel));
27    /* remove from n the nodes of black-listed choice branches */
28    foreach block b in blacklist do
29      N := N \  $\mathbb{N}$ (generateCFG(b));
30    end foreach
31    foreach node n' in N do
32      if (not n' dominates n && not n dominates n') then
33        P := P  $\cup$  actingParticipants(n');
34      end if
35    end foreach
36    /* generate one constraint per concurrently-acting participant */
37    foreach participant p in P do
38      res := res  $\circ$  [ new constraint  $\text{AW}_{\text{fired}}(n, p) = \text{ia}$  ];
39    end foreach
40  end if
41  return res;
42 }

```

Figure 4.10: Pseudo-code for deriving the awareness constraints attached to the *propagates exception* node *n*.

- the activity that generates the *propagates exception* node n is nested into a parallel activity, but n is encapsulated into a choreography nested in the parallel activity and, therefore, the exception is handled by that choreography’s exception handlers. (However, that nested choreography might have its own *propagates exception* nodes, which will be associated with awareness constraints if necessary.)

In both the cases described above, no awareness constraint is needed for n .

While traversing the hierarchy of parents of the activity that generates the *propagates exception* node looking for the outer parallel activity, the algorithm builds a *blacklist* of blocks. The blacklisted blocks are all the “other” branches of those choice activities into which is nested the activity generating the *propagates exception* node. That is, the blacklisted blocks encompass all those activities that *cannot* be executed concurrently to the activity generating the *propagates exception* node because they are mutually exclusive with it.

In case an outer parallel is found, the second part of the algorithm identifies which actions are interrupted the traversing of the *propagates exception* node n . These actions are all those represented by some node in the Control Flow Graph sub-graph of the parallel activity identified before, with the exception of the nodes that dominate by n (because they must have already been enacted before n is traversed) or those that are dominated by n (because they can be enacted only after n , and therefore they will *never* be enacted). Once all the actions that might be made non-enactable are found, the awareness constraints are generated by requiring that each acting participant of at least one of those interruptible actions is immediately aware of the firing of the event represented by n .

Consider again the example in Figure 4.10. The goal is to find which activities may be enacted in parallel by the throwing of the exception of type e_1 . The outer parallel activity is *prl*₁. The only blacklisted block is the one comprising **skip** [s_1], because it is the only block mutually exclusive with **throw** [t_1] e_1 . Therefore, the one action that can be enacted concurrently is the dispatching by p_2 of the message to type m_2 to p_1 . The participant p_2 is unfortunately unaware of the outcome of the decision by p_1 in the scope of **choice** [c_1], which means that the choreography in Figure 4.10 has a realizability defects due to the violation of the “Observe your actions becoming not enactable” realizability requirement.

4.4.2.2 Computational Complexity

The algorithm in Figure 4.10 is based on the traversing of the parsing tree of the choreography. Its computational complexity can be (very roughly) estimated as quadratic with respect to the number of activities in the choreography (i.e., the max depth of the parsing tree) times how many activities may be terminated by the enactment of a throw activity that, once again, we very conservatively estimate it to be less or equal to the number of activities specified by the choreography. We have already estimated the number of activities to be linear with the number of nodes in the CFG, and therefore this upper-bound can be over-approximated to:

$$\mathcal{O}(|\mathbb{N}(awm)|^2)$$

More precise and complex approximations are possible but seem ultimately not worth the effort: we are considering the computational upper-bound and this is not the highest among the various algorithms that compose the realizability analysis method (see Section 4.7).

4.5 On the Origin of Realizability Defect Types and Participant Awareness

The goal of this section is twofold. Firstly, Section 4.5.1 categorizes the types of realizability defects that can affect ChorTex choreographies and relate them with the realizability requirements introduced in Section 4.4. Secondly, Section 4.5.2 relates the types of realizability defects and

the definition of participant awareness with the constructs provided by a choreography modeling language.

4.5.1 Realizability Defect Types in ChorTex Choreographies

Section 4.4 has introduced three types of awareness constraints that, if verified on the Awareness Model generated from a ChorTex choreography, guarantee the strong realizability of the latter. When an awareness constraint is not satisfied on a choreography, it symptomizes a realizability defect. Indeed, the following categorization of the types of realizability defects is derived directly from the types of awareness constraints they are diagnosed by:

Realizability Defect Type 1 (Unawareness of non-initiating action enactability). In one or more of the enactments of the choreography, an acting participant of one non-initiating action is unaware of that action becoming enactable and therefore cannot perform it without running the risk of violating the choreography.

Realizability Defect Type 2 (Unawareness of initiating action enactability). In one or more of the enactments of the choreography, an acting participant of one initiating action is unaware of that action becoming action becomes enactable and therefore cannot perform it without running the risk of violating the choreography.

Realizability Defect Type 3 (Unawareness of action non-enactability). In one or more of the enactments of the choreography, an acting participant of one action is unaware of that action changing from enactable to non-enactable due to some exception propagation.

Realizability defects of Type 1 and Type 2 are violations of the “Observe your actions becoming enactable” realizability requirement and result from insufficient participant awareness by the acting participants on the enactability of their actions. We decided to “split” initiating and non-initiating actions in two different types of realizability defects due to the rather different requirements on participant awareness they imply (see Section 4.4.1.1 and Section 4.4.1.2). Realizability defects of Type 3 are violations of the “Observe your actions becoming not enactable” realizability requirement. In ChorTex, the only way an enactable action can become non-enactable is due to exception propagation.

4.5.2 From Choreography Modeling Constructs to Participant Awareness Dimensions

The types realizability defects discussed in Section 4.5 share a common theme: they are all caused by shortcomings in the participant awareness by a participant at some point of enactments of the choreography. The Conjecture of Harmful unawareness formalizes this intuition:

Conjecture (Harmful unawareness). Every realizability defect in a choreography is the result of modeling shortcomings that make one or more participants have in some enactments insufficient participant awareness to perform their role correctly.

In other words, every possible realizability shortcoming of a ChorTex choreography in terms of realizability has to do with a participant not having sufficient information on the state of the enactment to play its role as specified.

Notice, however, that the phrasing of the “Harmful unawareness” does not limit it to neither ChorTex choreographies nor the specific definition of realizability assumed in this work. The generality of the conjecture is specifically intended. (And, incidentally, it is the reason it is a conjecture and not a stronger form of statement: after all, we have shown that participant awareness is sufficient to characterize all realizability defects for ChorTex choreographies that require strong realizability.) In other words, does the Conjecture of Harmful unawareness apply to choreography modeling languages other than ChorTex and to definitions of realizability other than strong realizability? This is author strongly believes the answer to be the following:

Yes, depending on a revised notion of participant awareness that must be tailored to the modeling constructs provided by the adopted choreography modeling language and the definition of realizability analysis.

In Section 2.3.1.2 we have shown how definitions of realizability depend on choreography modeling languages (and therefore their constructs, which define the choreography modeling languages). Therefore, in the remainder of this discussion we will concentrate on the relation between modeling constructs and dimensions of participant awareness.

Participant awareness as defined in Section 4.2 is indeed explicitly tailored to the constructs of ChorTex. ChorTex is a choreography modeling language that focuses exclusively on the ordering of actions and not, for example, on the time elapsing between the execution of any two of them. Because of this focus on ordering of actions, participant awareness in ChorTex choreographies is *mono-dimensional*: it is limited to modeling the knowledge of the participants in terms of which actions are or are not enactable and which have already been enacted. Besides this dimension of “action enactability and enactment,” two other dimensions of participant awareness immediately come to mind when considering the state of the art of choreography modeling languages (see Section 2.2):

Messages content: in choreography modeling languages such as WS-CDL and BPMN v2.0 Choreography Diagrams, decisions can be modeled on the basis of the content of messages previously exchanged among the participants. Realizability defects arise in choreographies that are specified so that a participant, due to blindness, does not have access to the messages it needs to perform its decision.

Participant identities: some choreography modeling languages like WS-CDL, BPMN v2.0 Choreography Diagrams and several formalisms based on process algebras, support forms of “channel passing,” that is, the passing from one participant to another of a “reference” used to communicate with a third participant (see Section 2.2.1.1). Naturally, a realizability defect ensues in the case a participant with no reference to a second participant must send a message to the latter.

The “messages content” dimension of participant awareness does not apply to ChorTex because its typing system is *shallow*: message types are identified exclusively by their identifiers (see “Unequivocal message exchanges” design assumption, Section 3.1). As a result, decisions in ChorTex activities like choice and iteration are modeled as internal actions of the participants and do not explicitly tie their outcome to the message exchanges performed thus far in the enactment. ChorTex choreographies also do not require the “participant identities” dimension because all participants are known to all others “a priori,” meaning that, in order for a participant to dispatch a message to another, the only necessary information is the participant identifier of the latter (see the “Participants are known a-priori” Design Assumption).

From the previous discussion, it is clear that, given a choreography modeling language, there is a very strong link between its constructs and the dimensions of participant awareness that one must consider. This link seems actually to be one of *causal* nature:

Conjecture (Constructs imply participant awareness dimensions). If the operational semantics of some construct of a choreography modeling language is based on the knowledge of a certain type of information by the participants, there is a dimension of participant awareness related to that type of information that must be taken into account when verifying realizability. Additionally, there are types of realizability defects that may occur that are caused by lack, incompleteness or incorrectness of the knowledge of that information held by participants.

In other words, the “Constructs imply participant awareness dimensions” Conjecture states that, if some type of knowledge is integral to the operational semantics of constructs of a choreography modeling language, that type of knowledge will affect the realizability of choreographies using that choreography modeling language. It stands to reason that, if the behavior to be enacted

by the participants is reliant on them having some sort of knowledge, lack of that knowledge would jeopardize the enactment of the behavior.

To corroborate the “Constructs imply participant awareness dimensions” Conjecture, we will perform the following “thought experiment.” Consider the possibility of adding support for channel passing in ChorTex by introducing a class of participant identifiers that are not known “a priori,” therefore forsaking the “Participants are known a-priori” design assumption of ChorTex (see Section 3.1). When a participant that is already known sends a message to one that is not yet known, the identity of the latter is “bound” in the enactment. That is, that message has fundamentally acted as an *invitation* for its recipient. A participant receives a reference to another by either receiving a message from it (and implicitly associating the identifier of the sender with the endpoint from which the message was sent) or receiving from a third participant a special type of message containing one or more references to participants. Adding such forms of channel passing would require the extension of the realizability method presented in this chapter to handle the “participant identities” dimension of participant awareness. Because of the control-flow constructs offered by ChorTex, it seems relatively straightforward to verify that any participant has sufficient knowledge of the other participants by means of another algorithm based on Control Flow Graphs. More precisely, we are thinking of an adapted version of the classical reaching definitions algorithm (see, e.g., [123]) executed once for participant in the choreography: when a participant p receives a reference to another participant p' , it counts as an assignment to the variable p' from the perspective of p . If, by the time p must send a message to p' , there is no assignment on the variable p' from the perspective of p , that is the symptom of a realizability defect. Notably, as predicted by the “Constructs imply participant awareness dimensions” conjecture, this is a *new type* of realizability defect for ChorTex: none of those listed in Section 4.5.1 already cover it.

Additionally, another aspect to be taken into account with respect to the new channel passing is the possibility of “overbooking:” multiple participants concurrently invite different entities to play as a certain participant. Again, this is yet a new type of realizability defect, this time connected to conflicts in establishing the knowledge of the new participant’s identity. In terms of realizability analysis, this type of realizability defects could be dealt with using a modified version of the algorithm used in Section 4.4.2.1 to identify actions that could be concurrently enacted to the throwing of an exception: no two “invitation” message exchanges that have as recipient the same participant identifier can be concurrently enactable by different acting participants.

4.6 Unequivocal Enactments, Revisited

The “Unequivocal enactments” design assumption of ChorTex states that participants can map correctly incoming messages to the enactments in which they are dispatched (see Section 3.1). When stating the “Unequivocal enactments” design assumption, we claimed that this design assumption is easily realized by having participants include as meta-data “enactment identifiers” in the messages they dispatch. The enactment identifier is generated beforehand by the initiator, namely the participant that performs the first action in the enactment, effectively “kick-starting” the latter.

But how do we ensure that the enactment identifier is propagated so that every other participant, before dispatching a message, in the enactment, knows what the identifier of that enactment is? It turns out that, under the following assumption, strong realizability of a ChorTex choreography is sufficient but not necessary condition to guarantee that the enactment identifier is transmitted to all the acting participants that are not the initiator of the enactment by the time they perform their first action in the enactment:

Assumption (Propagation of enactment identifiers in opaque activities). After the completion of an opaque activity, all the participants therein involved have received the enactment identifier. Moreover, if the opaque activity is the initiating action of the enactment, over its execution its participants have collectively generated exactly one enactment identifier.

Notice that this assumption is consistent with the assumption that all participants that are invoked in an opaque activity are eventually aware of its completion (see Section 4.3.3). The proofs

that strong realizability is a sufficient but not necessary condition for unequivocal enactments are presented in Section 4.6.1 and Section 4.6.2, respectively.

4.6.1 Strong Realizability implies Unequivocal Enactments

The proof of this claim is a direct consequence of the following lemmas:

Lemma (Flow of participant awareness). When enacting a strongly realizable choreography, a participant p that is not the initiator is aware of an action a becoming enactable, then p has observed an action a' being executed and there is a path, possibly of length zero, made exclusively of reactive nodes that connect the participant-activated nodes n_a and n'_a that represent a and a' , respectively.

Proof. The proof of this lemma is based directly on how the participant awareness is calculated: the Awareness Annotation Algorithm is a fixed-point algorithm defined on Control Flow Graphs. The function \boxplus , defined in Section 4.3.2, specifies how the participant awareness of a participant p of a node n becoming enactable is based on the participant awareness of the enactment of the predecessor nodes of n . Specifically, p is eventually aware of n becoming enactable if and only if p is immediately or eventually aware of the enactment of all the predecessors of n . How the traversing of the node n affects the participant awareness of the participant p is specified by the function $f(n, p)$ that has been defined in Section 4.3.3. The $f(n, p)$ is defined so that the only possibility for p being eventually aware after traversing a reactive node n is that p was already eventually aware of n being enactable. Therefore, if p is eventually aware of the enactment of all the reactive nodes of a path, p must have also been eventually aware of the first of those nodes becoming enactable. Assume that the predecessor of the first node of the path is a participant-activated node. Because of the definition of $f(n, p)$, if p is to be eventually aware of its enactment of n , then p must have observed its enactment it. \square

Lemma (Well-rooted actions). When enacting a strongly realizable choreography, the sender of a message exchange or every participants of an opaque activity is either the initiating participant or has necessarily already received a message or participated in an opaque activity.

Proof. This proof builds directly on top of the “Flow of participant awareness” lemma. Let us consider an arbitrary message exchange activity a that is not the first one executed in an enactment. If a is an initiating action, then its acting participants must be aware of the initial action been performed: the choreography is strongly realizable, and therefore all the Type 2 awareness constraints are satisfied and particularly those associated with the participant-activated node representing a . Since the acting participants of a are aware of the performing of the first action in the enactment, the “Well-rooted actions” lemma is satisfied for a .

Now, let us consider the other case, namely that a is not an initiating action. Since a is not an initiating action, it has one or more Type 1 awareness constraints associated with it. Those Type 1 awareness constraints must be satisfied, because the choreography is strongly realizable. Therefore, because of the “Flow of participant awareness” lemma, the acting participants of a that are not the initiator must have already observed another action a' in the enactment. If a' is a message exchange action or an opaque action, then the “Well-rooted actions” lemma is verified. Otherwise, a' is a decision in the scope of a choice or an iteration activity. In this case, we can apply the “Well-rooted actions” lemma on a' and use its obvious transitivity. The only way in which “Well-rooted actions” lemma would not be verified is if the only actions observed by a participant that is not the initiator were all decisions in choice or iteration activities. (If p is the initiator, the “Well-rooted actions” lemma is already satisfied.) We show, proceeding by absurd, that such a scenario contradicts the assumption that the choreography is strongly realizable. Let us assume that a' is a decision and that also every other action that p has observed so far is also a decision. A participant can observe a decision only if it is the decision maker (see Definition 4.4). Therefore, all decisions observed by p were taken by p . We consider the following two cases:

- At least one of the actions of p is an initiating action;

- None of the actions of p are initiating actions.

In the first case, the contradiction is the following. Since the choreography is realizable, all Type 2 awareness constraints are satisfied. Therefore, if any of the decisions performed by p are initiating actions, all the initiating actions of the choreography *must be* decisions taken by p (any other possibility contradicts the fact that there are no Type 2 realizability defects). But since p is the acting participant of all initiating actions, he is also necessarily the initiator, which leads to a contradiction.

In the second case, the contradiction stems from the “Flow of participant awareness” lemma. Given the first decision performed by p is not an initiating action, the fact that the choreography is strongly realizable guarantees that there are no Type 1 realizability defects associated with it. This implies that p must be immediately or eventually aware of the the decision becoming enactable. However, the “Flow of participant awareness” lemma requires the existence of another action that p has observed before performing the decision, which contradicts the assumption that the decision is the first action observed by p . \square

4.6.2 Unequivocal Enactments do not Require Strong Realizability

The fact that strong realizability is not a necessary condition for unequivocal enactments is straightforwardly proven with a counterexample, namely a choreography that is not strongly realizable, but in which enactment identifiers are correctly propagated.

Proof. Consider the following choreography:

```

1 chor [chor0] ({
2   [mex0]  $p_1 \rightarrow m_0$  to  $p_2, p_3$ ;
3   [mex1]  $p_2 \rightarrow m_1$  to  $p_1$ ;
4   [mex2]  $p_3 \rightarrow m_2$  to  $p_2$ 
5 })
```

The enactment identifier is generated by p_1 and delivered to p_2 and p_3 over the message m_0 . Therefore, both p_2 and p_3 have received the enactment identifier by the time they dispatch the messages m_1 and m_2 , respectively. However, the choreography in the example is not strongly realizable: there is a Type 1 realizability defect associated with the node representing the dispatching of m_2 because p_3 is unaware of the dispatching of the message m_1 . \square

4.7 Discussion

The concept of awareness has already been applied in the state of the art to the end of realizability analysis methods (see Section 2.3.4.1). What is novel in our approach is the *granularity* of participant awareness. All other works we know of treat participant awareness on a “yes/no” basis. In our realizability analysis method, the “yes” case of participant awareness is further differentiated between immediate- and eventual-awareness and the “no” one in unawareness and non-involvement. The difference between immediate- and eventual-awareness allows us to differentiate between synchronous internal actions, namely decisions and dispatching of messages by participants, and asynchronous ones like the reception of messages and the participation in opaque activities.

Another novelty of our approach is the pervasive use of Control Flow Graphs and associated analysis techniques. Building directly on top of such very well-known concepts grants us access to a rich set of algorithms, the performance of which has been honed sharp by decades of research. (Not to mention the fact that implementations of the entire Control Flow Graph framework and associated algorithms are freely available, such as the one we reuse in the prototype, see Chapter 6.) A detailed discussion of the overall computational complexity of our realizability analysis method is presented in Section 4.7.1.

In Section 4.7.2, instead, we reconsider Type 3 realizability defects and argue that constructs that may cause such defects should best be avoided or, at the very least, considered with extreme care when creating new choreography modeling languages.

4.7.1 Overall Upper-Bound Computational Complexity

The diagram in Figure 4.11 combines the information we have on the computational complexity of the various pieces of the realizability analysis method.

The complexity of generating a Control Flow Graph from a choreography is linear with respect to the number of activities therein defined (see Section 3.4). As discussed in Section 4.3.4, the amount of nodes in the Control Flow Graph is comparable to (but strictly bigger than) the amount of activities specified in the choreography.

The upper-bound complexity of the Awareness Annotation Algorithm has been estimated in Section 4.3.4 as the following:

$$\mathcal{O}(|\mathbb{N}(awm)|^3 \times |\mathbb{P}(chor)|)$$

The notation $|\mathbb{N}(awm)|$ denotes the amount of nodes in the Awareness Model awm . Since the upper-bound complexity of the Awareness Annotation Algorithm is higher than those of all the other algorithms employed in the realizability analysis, it can be assumed to be the overall upper-bound complexity for generating Awareness Models.

Once an Awareness Model is generated, the evaluation of awareness constraints is mostly a matter of looking up a pre-calculated value annotated on the Awareness Model nodes; therefore, we can estimate its upper-bound complexity to a negligible $\mathcal{O}(1)$. The generation of the awareness constraints, however, is significant in terms of computational complexity and depends on the type of awareness constraints taken into account. In the case of Type 1 and Type 2 awareness constraints, it is a matter of scanning all the nodes of the Awareness Model and looking up their acting participants; therefore, it has an upper-bound computational complexity of $\mathcal{O}(|\mathbb{N}(awm)|)$. The upper-bound complexity of the algorithm to generate Type 3 awareness constraints has already been estimated in Section 4.4.2.2 to:

$$\mathcal{O}(|\mathbb{N}(awm)|^2)$$

Combining the upper-bound computational complexity of all its parts, the overall upper-bound computational complexity of the realizability analysis presented in this chapter is shown in Figure 4.12. The final outcome is approximated keeping into account that we are looking for the upper-bound complexity (hence only the “biggest” term in the sum really counts), and that the number of activities in the choreography is always smaller than the nodes in the respective Awareness Model (as discussed earlier in this section).

As argued in Section 2.3.3, it makes little sense to compare the raw performance (e.g., in term of execution time) of realizability analysis methods devised for different choreography modeling, in particular when different definitions of realizability are being checked. Computational complexity, on the other hand, gives a rather concrete feeling for the feasibility of a particular realizability analysis method. The computational complexity of our method compares extremely favorably with the methods based on automata [102, 116, 157, 109] or Petri-Nets [81, 135], which have to resort to PSPACE-complete language-equivalence checks (see also [134]). The reason for the very low computational complexity of our realizability analysis method is the adoption of the concept of participant awareness. To draw a comparison with model-checking, verifying awareness constraints is fundamentally *symbolic model checking* for realizability. Intuitively, eventual awareness is an “expedient” to reduce the size of the state-space of the realizability problem by *hiding* the actions of receiving messages performed by the recipients. One should therefore not be surprised that the computational complexity is far lower than *explicit-state model checking* methods for realizability analysis like the constructive ones listed in Section 2.3.3.

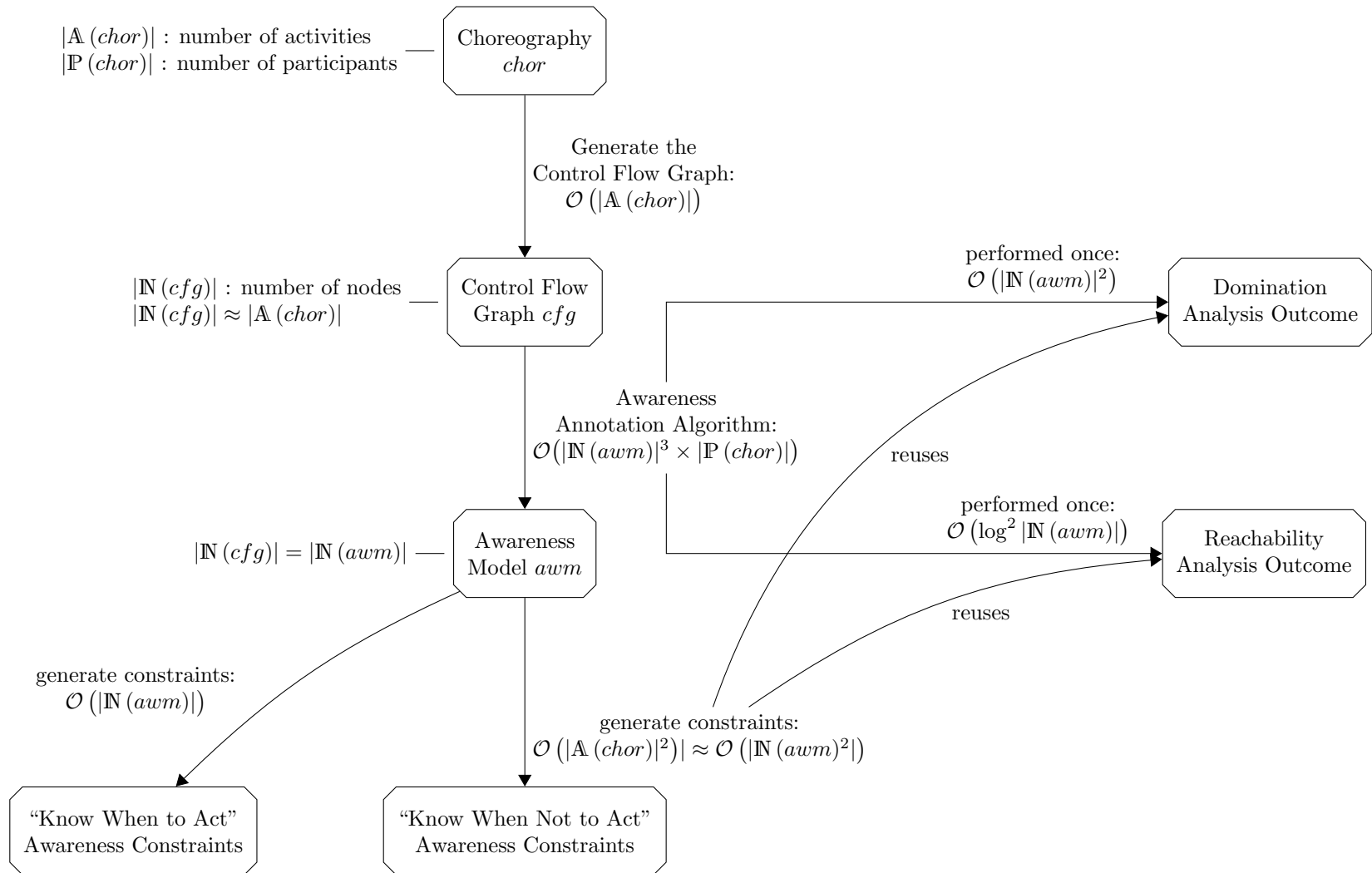


Figure 4.11: Diagram that summarizes the awareness-based realizability analysis and correlates the upper-bound complexity computational complexity of its different parts.

$$\begin{aligned}
& \mathcal{O}(|\mathbb{A}(chor)|) + && \text{(Generation Control Flow Graph)} \\
& \mathcal{O}(|\mathbb{N}(awm)|^2) + && \text{(Domination Analysis)} \\
& \mathcal{O}(\log^2 |\mathbb{N}(awm)|) + && \text{(Reachability Analysis)} \\
& \mathcal{O}(|\mathbb{N}(awm)|^3 \times |\mathbb{P}(chor)|) + && \text{(Awareness Annotation Algorithm)} \\
& \mathcal{O}(|\mathbb{N}(awm)|) + && \text{(Generation “know when to act” constraints)} \\
& \mathcal{O}(|\mathbb{A}(chor)|^2) = && \text{(Generation “know when not to act” constraints)}
\end{aligned}$$

$$\mathcal{O}(|\mathbb{N}(awm)|^3 \times |\mathbb{P}(chor)|) \qquad \text{(Overall Upper-Bound Complexity)}$$

Figure 4.12: Overall upper-bound computational complexity of the strong realizability analysis of ChorTex choreographies.

4.7.2 A Case Against Activity-Terminating Exception Handling (and on the Design of Choreography Modeling Languages)

The complexity and subtlety of Type 3 realizability defects make a compelling case against the (wanton) adoption of *terminating constructs* in choreography modeling languages that assume asynchronous communication models.

Terminating constructs are those, such as the exception throwing mechanism of ChorTex, that cause the immediate interruption of concurrently executing activities. Terminating constructs are very hard to handle both in terms of realizability analysis methods as well as in terms of modeling practice. In Section 4.4.2, we have been able to exploit the block-based structure of ChorTex and the relatively small set of constructs it provides to identify the actions that could be executed concurrently with the throwing of an exception. Similar “neat tricks” are definitely unlikely in the scope on graph-based modeling languages like BPMN v2.0 Choreography Diagrams that employ Petri net-like operational semantics based on tokens. Due to the graph-based structure, the only reliable way to identify actions that can occur in parallel seems to be a full-fledged state-space search, which is exponential [95] rather than polynomial like in ChorTex. Therefore, it is very likely that realizability analysis methods with exponential complexity are simply too slow to be implemented in IDEs and run continuously in the background while the modeler models the choreography.

Besides the computational complexity of the verification, terminating constructs require draconian restrictions to the rest of the choreography in order for the latter to be strongly realizable. As discussed in Section 4.4.2, the combination of exception throwing and the asynchronous communication model assumed in ChorTex requires that fundamentally one single participant is “doing all the work” in the parts of the choreography that are susceptible to termination. Indeed, exception throwing is “safe” from a realizability perspective only if, whenever it may occur, the actions that may be concurrently performed have the same, unique acting participant. An intuitive proof is based on the observation that at most one acting participant is immediately aware after the performing of any one action (see Table 4.2). Since all participants that may act concurrently to the throwing of an exception must be immediately aware of the latter, and that only one can be so at any time, it follows that there can be only one unique acting participant in parts of strongly realizable choreographies that can be terminated by the throwing or propagation of exceptions.

These issues with terminating constructs are not a phenomenon isolated to ChorTex: they are simply a *harsh reality of distributed systems*. For example, BPMN v2.0 allows to specify **terminate end event** (see Section 2.2.2.4), which have the same effect of the throwing of an exception that terminates an entire ChorTex choreography. With respect to **terminate end events** in BPMN v2.0 choreographies, the BPMN v2.0 Choreography specification [163, p. 344] contains the following, extremely interesting excerpt:

“[Terminate end events can be used in a Choreography, however] there would be no specific ability to terminate the Choreography, since there is no controlling system. In this case, all Participants in the Choreography would understand that when the Terminate End Event is reached (actually when the Message that precedes it occurs), then no further messages will be expected in the Choreography, even if there were parallel paths. The use of the Terminate End Event really only works when there are only two Participants. If there are more than two Participants, then any Participant that was not involved in the last Choreography Task would not necessarily know that the Terminate End Event had been reached.”

This quote seems to contain a factual error: similarly to what happens with ChorTex choreographies, Terminate End Events are almost certainly usable in BPMN v2.0 choreographies using asynchronous communication model with any amount of participants, as long as only one of them can act when a Terminate End Event can occur. The fact that even a specification like BPMN v2.0 is affected by such misconceptions is in itself a statement to the complexity and subtlety of Type 3 realizability defects. (Moreover, if even experts get that wrong, we can only expect end users to

be little short of befuddled.) Nevertheless, the BPMN v2.0 specification itself argues about strong limitations when terminating constructs are used in choreographies.

More in general, as noted in [121], the distributed nature of service choreographies requires a radical re-thinking of how error handling, and terminating constructs like exception throwing in ChorTex or terminating end events in BPMN v2.0 Choreography Diagrams *are definitely not the right solution*. Exception handling and termination events are constructs that work well in service orchestrations, where all the activities are executed by the same actor: the orchestration engine. Given the close relationship between service orchestrations and choreographies, we understand very well how tempting it is to “port” service orchestration constructs to service choreographies. After all, similar constructs in both service orchestrations and choreographies intuitively simplifies the projection of the roles (see, e.g., [205]) and softens the learning curve for the modelers.

The slogan of a world-class manufacturer of tires goes, “power is nothing without control.” And, in this author’s opinion, terminating constructs are just a tip of the following, dreadful iceberg: we have sacrificed a lot of control in terms of modeling and verifying choreographies for the sake of having extremely powerful (as in expressive) orchestration languages. In other words, as a community we may have gotten wrong the “direction” of the porting: instead of adding to service choreographies constructs that are known to work in service orchestrations, we should remove from service orchestrations constructs that are known *not to work* in service choreographies! The following approach is very likely to bear good results for the design of well-integrated choreography and orchestration languages. One should start from the “hardest” of the two domains: choreographies. The choreography modeling language should be defined to include only those constructs that are known to be manageable in the scope of choreography realizability as well as compliance and other verification problems [191]. Constructs that are manageable in choreographies are almost certainly so also in orchestrations. This way, we would sacrifice expressiveness and maybe some usability of the orchestration modeling language, but we would gain manageability and verifiability of the choreography one. Given the notoriously high costs for correcting defects in distributed systems, the trade seems absolutely a bargain.

Chapter 5

Realizability-Driven Evolution of ChorTex Choreographies

In Chapter 4 is presented a method to analyze the strong realizability of ChorTex choreographies and to diagnose the realizability defects affecting them. It has been said that “knowing is half the battle;”¹ presumably, the other half of the battle is acting upon one’s knowledge. Given the ultimate goal of this thesis to facilitate the modeling-time specification of realizable interaction choreographies (see Section 1.2), it means that the knowledge of their realizability defects affecting a ChorTex choreography must be complemented by effective means of correcting them. The present chapter aims at accomplishing this by presenting several *remediation strategies*. A remediation strategy is an algorithm to automatically calculate *remediation plans* for an unrealizable choreography, i.e., modifications to be applied to that choreography in order to solve one or more of its realizability defects.

The first step towards remediation plans is the definition of the *change algebra* presented in Section 5.1, namely a set of *change operators* that enable the modification of ChorTex choreographies. The desiderata and design rationale of our framework of remediation strategies are laid out in Section 5.2. Section 5.3 discusses how changing a choreography can result in the introduction of new realizability defects and defines invariants to prevent it. Section 5.4 presents the remediation strategies we propose grouped by the type of realizability defect they address. The remediation plans generated with our remediation strategies tackle single realizability defects. In case a choreography is affected by multiple realizability defects, modelers have to decide which realizability defect to tackle first; guidelines to this end are proposed in Section 5.5.

5.1 A Change Algebra for ChorTex

A change algebra is a set of change operators (sometimes also called “change patterns” [198]) that act as primitives to modify models. In our case, we are interested in change operators for modifying ChorTex choreographies.

Definition 5.1 (Change Operators). A change operator c is a function that takes in input the well-formed specification of the ChorTex activity \mathbf{A} and additional arguments $\alpha_1, \dots, \alpha_n$, the type and number of which depends on the particular change operator, and yields as output the well-formed activity specification \mathbf{A}' . A change operator may specify any number of pre-conditions on its input to guarantee that the output is well-formed.

The change operators defined in the remainder of this section assume that the choreographies they are applied to are well-formed (see Section 3.5) and are specified so that their application to

¹The fact that this citation comes from the G.I. Joe public service announcements does not invalidate in the least the fundamental truth it conveys. (Just ask Cobra commander.)

a well-formed choreography results in another well-formed choreography. While the preservation of well-formedness is not generally a requirement for change operators, it is extremely convenient in our case. The realizability analysis presented in Chapter 4 is applicable only to well-formed ChorTex choreographies (see, in particular, Section 4.1). Having change operators that preserve well-formedness enables us to execute the realizability analysis immediately after the application of any amount of changes to a choreography, which is extremely important both in the scope of the framework of remediation plans and the guidelines for dealing with multiple realizability defects discussed in Section 5.2 and Section 5.5, respectively.

Figure 5.1 provides an outlook of the ChorTex change algebra. Its change operators are grouped in the following three categories:

Insertion change operators yield changes that modify the choreography by adding new activities to it (Section 5.1.1);

Update change operators yield changes that modify *information items* of existing activities (Section 5.1.2);

Deletion change operators yield changes that modify the choreography removing activities from it (Section 5.1.3).

Information items are defined in [31, 32] as “atomic information.” The information items in ChorTex choreographies are specified in the ChorTex syntax as terminal symbols of type **ID**, which are used to denote activity names, participant identifiers, message types and exception types (see Section 3.2).

The remainder of this section treats each change operator separately. For the sake of the formalization of the change operators, in the remainder it is assumed a root choreography named *chor*. Preconditions that are specified to ensure well-formedness of the output activity are labelled with the respective Well-formedness Requirement (see Section 3.5). Finally, Section 5.1.4 briefly discusses how to achieve modifications that are not addressed by single change operators by combining them.

5.1.1 Insertion Change Operators

Insertion change operators enable the addition of new activities to the choreography.

5.1.1.1 Insert Activity in a Block

This change operator allows to “inject” an activity at a specified position in an already-existing block. The activity to be nested into the block may be of any type, including a choreography, as long as it is well-formed.

Input 1: A well-formed block:

$$\{\mathbf{A}_1; \dots; \mathbf{A}_n\}$$

Input 2: A well-formed activity **A**

Input 3: An index i so that: $1 \leq i \leq n + 1$

Precondition (Unique Activity Names): None of the names specified by **A** and its nested activities (if any) clashes with the name of any other activity in the root choreography *chor*. Formally:

$$names(\mathbf{A}) \cap names(chor) = \emptyset$$

The function $names(\mathbf{A})$ returns the set of names declared by the activity **A** and its nested activities (see Section 3.5).

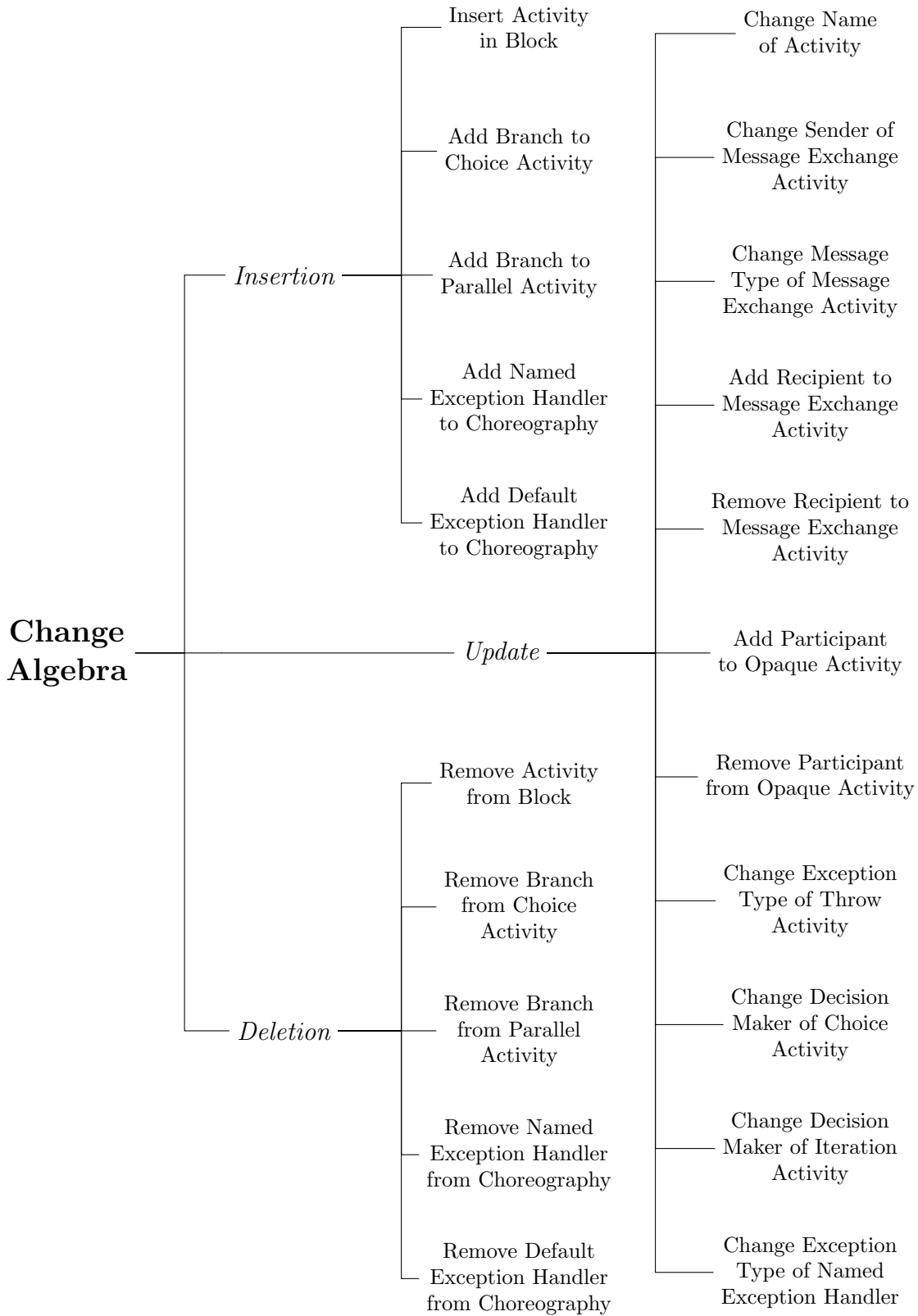


Figure 5.1: Outlook of the ChorTex change algebra; the change operators are the leaves on the right side of the tree; the categories are typeset in italics.

Precondition (Unique Message Types): None of the message types specified by \mathbf{A} and its nested activities (if any) clashes with the message types specified by any other activity in the root choreography $chor$. Formally:

$$msgTypes(\mathbf{A}) \cap msgTypes(chor) = \emptyset$$

The function $msgTypes(\mathbf{A})$ returns the set of message types declared by the activity \mathbf{A} and its nested activities (see Section 3.5).

Output: The well-formed block:

$$\{\mathbf{A}_1; \dots; \mathbf{A}_{i-1}; \mathbf{A}; \mathbf{A}_i; \dots; \mathbf{A}_n\}$$

5.1.1.2 Add Branch to Choice Activity

This change operator allows the addition of a new branch to an existing choice activity.

Input 1: A well-formed choice activity:

$$\text{choice } p \text{ either } \mathbf{A}_1 \text{ or } \dots \text{ or } \mathbf{A}_n$$

Input 2: A well-formed block \mathbf{A}

Input 3: An index i so that: $1 \leq i \leq n + 1$

Precondition (Unique Activity Names): None of the names specified by \mathbf{A} and its nested activities (if any) clashes with the name of any other activity in the root choreography $chor$. Formally:

$$names(\mathbf{A}) \cap names(chor) = \emptyset$$

Precondition (Unique Message Types): None of the message types specified by \mathbf{A} and its nested activities (if any) clashes with the message types specified by any other activity in the root choreography $chor$. Formally:

$$msgTypes(\mathbf{A}) \cap msgTypes(chor) = \emptyset$$

Output: The well-formed choice activity:

$$\text{choice } p \text{ either } \mathbf{A}_1 \text{ or } \dots \text{ or } \mathbf{A}_{i-1} \text{ or } \mathbf{A} \text{ or } \mathbf{A}_i \text{ or } \dots \text{ or } \mathbf{A}_n$$

5.1.1.3 Add Branch to Parallel Activity

This change operator allows the addition of a new branch to an existing parallel activity.

Input 1: A well-formed parallel activity:

$$\text{parallel do } \mathbf{A}_1 \text{ and } \dots \text{ and } \mathbf{A}_n$$

Input 2: A well-formed block \mathbf{A}

Input 3: An index i so that: $1 \leq i \leq n + 1$

Precondition (Unique Activity Names): None of the names specified by \mathbf{A} and its nested activities (if any) clashes with the name of any other activity in the root choreography $chor$. Formally:

$$names(\mathbf{A}) \cap names(chor) = \emptyset$$

Precondition (Unique Message Types): None of the message types specified by \mathbf{A} and its nested activities (if any) clashes with the message types specified by any other activity in the root choreography $chor$. Formally:

$$msgTypes(\mathbf{A}) \cap msgTypes(chor) = \emptyset$$

Precondition (Univocal Exception Triggering): Consider the two Control Flow Graphs c_p and c_b generated by the input parallel activity and the input block, respectively. If any exception propagation node is reachable from the start node in c_p through a path made exclusively of reactive nodes, then there must be no exception propagation node reachable from the start node in c_b through a path made exclusively of reactive nodes.

Output: The well-formed parallel activity:

parallel do \mathbf{A}_1 and ... and \mathbf{A}_{i-1} and \mathbf{A} and \mathbf{A}_i and ... and \mathbf{A}_n

The precondition for enforcing the “Univocal Exception Triggering” well-formedness requirement is by far the most complicated to be found in the change operators presented in this work. Its complexity comes from the fact that the well-formedness requirement it aims at enforcing is specified *indirectly* on the structure of ChorTex choreographies by means of predicates on their Control Flow Graphs. The goal of this precondition is to avoid that, by inserting a branch to a parallel activity, it could happen that two exceptions are thrown concurrently when enacting the output parallel activity, which is not supported by the operational semantics of ChorTex and violates the “Univocal Exception Triggering” well-formedness requirement (see Section 3.5). Recall that paths in a Control Flow Graphs containing only reactive nodes are traversed instantly as soon as their source node is reached (Section 3.4.2). In terms of Control Flow Graph, two exceptions would be thrown concurrently as soon as the output parallel activity is initiated if, from the start node of its Control Flow Graph, would originate two separate paths made only of reactive nodes and ending in exception propagation nodes. Therefore, if the input parallel activity already has one exception propagation node reachable from the start node through a path made exclusively of reactive nodes, then the block activity that will be inserted into it must not, as it would add *another exception propagation node* reachable from the start node through a *different* path made only of reactive nodes.

5.1.1.4 Add Named Exception Handler to Choreography

This change operator allows the addition of a new named exception handler to an existing choreography.

Input 1: A well-formed choreography:

$$chor(\mathbf{A} \mid e_1: \mathbf{A}_{e_1} \mid \dots \mid e_n: \mathbf{A}_{e_n} \mid *: \mathbf{A}_*)$$

Input 2: An exception type e

Input 3: A well-formed block \mathbf{A}_e

Input 4: An index i so that: $1 \leq i \leq n + 1$

Precondition (Unique Activity Names): None of the names specified by \mathbf{A} and its nested activities (if any) clashes with the name of any other activity in the root choreography $chor$. Formally:

$$names(\mathbf{A}_e) \cap names(chor) = \emptyset$$

Precondition (Unique Message Types): None of the message types specified by \mathbf{A} and its nested activities (if any) clashes with the message types specified by any other activity in the root choreography $chor$. Formally:

$$msgTypes(\mathbf{A}_e) \cap msgTypes(chor) = \emptyset$$

Precondition (Unique Named Exception Handler Types): None of the named exception handlers already declared the input choreography catches e . Formally:

$$e \notin \{e_1, \dots, e_n\}$$

Output: The well-formed choreography:

$$chor(\mathbf{A} | e_1 : \mathbf{A}_{e_1} | \dots | e_{i-1} : \mathbf{A}_{e_{i-1}} | e : \mathbf{A}_e | e_i : \mathbf{A}_{e_i} | \dots | e_n : \mathbf{A}_{e_n} | * : \mathbf{A}_*)$$

5.1.1.5 Add Default Exception Handler to Choreography

This change operator allows the addition of a default exception handler to an existing choreography that does not yet have one. The new default exception handler is specified by means of the block that will constitute its body.

Input 1: A well-formed choreography that does not specify a default exception handler:

$$chor(\mathbf{A} | e_1 : \mathbf{A}_{e_1} | \dots | e_n : \mathbf{A}_{e_n})$$

Input 2: A well-formed block \mathbf{A}_*

Precondition (Unique Activity Names): None of the names specified by \mathbf{A} and its nested activities (if any) clashes with the name of any other activity in the root choreography $chor$. Formally:

$$names(\mathbf{A}_*) \cap names(chor) = \emptyset$$

Precondition (Unique Message Types): None of the message types specified by \mathbf{A} and its nested activities (if any) clashes with the message types specified by any other activity in the root choreography $chor$. Formally:

$$msgTypes(\mathbf{A}_*) \cap msgTypes(chor) = \emptyset$$

Output: The well-formed choreography:

$$chor(\mathbf{A} | e_1 : \mathbf{A}_{e_1} | \dots | e_n : \mathbf{A}_{e_n} | * : \mathbf{A}_*)$$

5.1.2 Update Change Operators

Update change operators allow to modify parameters specified in activities, e.g., their name, the types of exceptions thrown, the participants that send or receive a certain message or those that take part in an opaque activity.

5.1.2.1 Change Name of Activity

This change operator allows the modification of the name associated with an existing activity.

Input 1: A well-formed activity \mathbf{A} with name id

Input 2: A name id' so that $id \neq id'$

Precondition (Unique Activity Names): The new name id' does not clash with the name of any other activity in the root choreography $chor$. Formally:

$$id' \notin names(chor)$$

Output: The well-formed activity \mathbf{A} , now associated with the name id' instead of id .

5.1.2.2 Change Sender of Message Exchange Activity

This change operator allows the modification of the sender of an existing message exchange activity.

Input 1: A well-formed message exchange activity:

$$p \rightarrow m \text{ to } p_1, \dots, p_n$$

Input 2: A participant identifier p'

Precondition (Sender is not a Recipient): The participant p' is not already a recipient in the message exchange activity. Formally:

$$p' \notin \{p_1, \dots, p_n\}$$

Output: The well-formed message exchange activity:

$$p' \rightarrow m \text{ to } p_1, \dots, p_n$$

5.1.2.3 Change Message Type of Message Exchange Activity

This change operator allows the modification of the type of the message exchanged over an existing message exchange activity.

Input 1: A well-formed message exchange activity:

$$p \rightarrow m \text{ to } p_1, \dots, p_n$$

Input 2: A message type identifier m'

Precondition (Unique Message Types): The message type m' does not clash with any of the message types specified by any other activity in the root choreography $chor$. Formally:

$$m' \notin msgTypes(chor)$$

Output: The well-formed message exchange activity:

$$p \rightarrow m' \text{ to } p_1, \dots, p_n$$

5.1.2.4 Add Recipient to Message Exchange Activity

This change operator allows the addition of a recipient to an existing message exchange activity.

Input 1: A well-formed message exchange activity:

$$p \rightarrow m \text{ to } p_1, \dots, p_n$$

Input 2: A participant identifier p'

Precondition (Distinct Recipients in Message Exchange Activities): The participant identifier p' must not already appear in the list of recipients of the message exchange activity. Formally:

$$p' \notin \{p, p_1, \dots, p_n\}$$

Output: The well-formed message exchange activity:

$$p \rightarrow m \text{ to } p_1, \dots, p_n, p'$$

Since the order of the recipients does not influence the operational semantics of a message exchange activity, for the sake of simplicity this change operator does not support the specification of an index at which the new recipient is inserted.

5.1.2.5 Remove Recipient from Message Exchange Activity

This change operator allows the removal of a recipient from an existing message exchange activity.

Input 1: A well-formed message exchange activity:

$$p \rightarrow m \text{ to } p_1, \dots, p_n$$

Input 2: A participant identifier p' so that $p' \in \{p_1, \dots, p_n\}$

Precondition (Syntactic Correctness): Message exchange activities must declare at least one recipient. To prevent the output message exchange activity from violating the syntax of ChorTex, the input message exchange activity specifies at least two recipients:

$$n \geq 1$$

Output: The well-formed message exchange activity:

$$p \rightarrow m \text{ to } p'_1, \dots, p'_{n-1}$$

The recipients in the message exchange activity in output are those of the activity in input minus the participant p' , that is:

$$\{p'_1, \dots, p'_{n-1}\} := \{p_1, \dots, p_n\} \setminus \{p'\}$$

5.1.2.6 Add Participant to Opaque Activity

This change operator allows the addition of a participant to an existing opaque activity.

Input 1: A well-formed opaque activity:

$$\text{opaque } (p_1, \dots, p_n)$$

Input 2: A participant identifier p' so that $p' \notin \{p_1, \dots, p_n\}$

Precondition (Distinct Participants in Opaque Activities): The participant identifier p' must not already appear in the list of participants of the opaque activity. Formally:

$$p' \notin \{p_1, \dots, p_n\}$$

Output: The well-formed activity:

$$\text{opaque } (p_1, \dots, p_n, p)$$

Similar to the case of the recipients of a message exchange activity, the order of the participants of an opaque activity does not influence its operational semantics. Therefore, for the sake of simplicity this change operator does not support the specification of an index at which the new recipient is inserted.

5.1.2.7 Remove Participant from Opaque Activity

This change operator allows the removal of a recipient from an existing opaque activity.

Input 1: A well-formed opaque activity:

$$\text{opaque } (p_1, \dots, p_n)$$

Input 2: A participant identifier p' so that $p' \in \{p_1, \dots, p_n\}$

Precondition (Syntactic Correctness): Opaque activities must declare at least two participants. To prevent the output opaque activity from violating the syntax of ChorTex, the input opaque activity must specify at least three participants:

$$n \geq 3$$

Output: The well-formed opaque activity:

$$\text{opaque } (p'_1, \dots, p'_{n-1})$$

The participants of the opaque activity in output are those of the opaque activity in input, minus the participant p' , namely:

$$\{p'_1, \dots, p'_{n-1}\} := \{p_1, \dots, p_n\} \setminus \{p'\}$$

5.1.2.8 Change Exception Type of Throw Activity

This change operator allows the modification of the type of the exception thrown by an existing throw activity.

Input 1: A well-formed throw activity:

$$\text{throw } e$$

Input 2: A name e' of an exception type so that $e' \neq e$

Output: The well-formed throw activity:

$$\text{throw } e'$$

5.1.2.9 Change Exception Type of Named Exception Handler

This change operator allows the modification of the type of the exception caught by an existing named exception handler specified by a choreography.

Input 1: A well-formed choreography that specifies at least a named exception handler:

$$\text{chor } (\mathbf{A}_b \mid e_1: \mathbf{A}_{e_1} \mid \dots \mid e_n: \mathbf{A}_{e_n} \mid *: \mathbf{A}_*)$$

Input 2: An exception type identifier e that is associated with a named exception handler in the choreography, namely:

$$e \in \{e_1, \dots, e_n\}$$

Input 3: An exception type identifier e'

Precondition (Unique Named Exception Handler Types): The exception type e' is not already caught by a named exception handler in the input choreography. Formally:

$$e' \notin \{e_1, \dots, e_n\}$$

Output: The well-formed choreography activity:

$$\text{chor } (\mathbf{A}_b \mid e'_1: \mathbf{A}_{e'_1} \mid \dots \mid e'_{n+1}: \mathbf{A}_{e'_{n+1}} \mid *: \mathbf{A}_*)$$

The participants of the opaque activity in output are those of the opaque activity in input, minus the participant p' :

$$\{e'_1, \dots, e'_{n+1}\} := \{e'\} \cup \{e_1, \dots, e_n\} \setminus \{e\}$$

For simplicity, the specification of this change operator assumes that the input choreography has a default exception handler. However, this assumption is meant only to simplify the formal presentation and has no real implication on the operational semantics of this change operator.

5.1.2.10 Change Decision Maker of Choice Activity

This change operator allows the modification of the decision maker of an existing choice activity.

Input 1: A well-formed choice activity:

$$\text{choice } p \text{ either } \mathbf{A}_1 \text{ or } \dots \text{ or } \mathbf{A}_n$$

Input 2: A participant identifier p' so that $p' \neq p$

Output: The well-formed choice activity:

$$\text{choice } p' \text{ either } \mathbf{A}_1 \text{ or } \dots \text{ or } \mathbf{A}_n$$

5.1.2.11 Change Decision Maker of Iteration Activity

This change operator allows the modification of the decision maker of an existing iteration activity.

Input 1: A well-formed iteration activity:

$$\text{iteration } p \text{ do } \mathbf{A}$$

Input 2: A participant identifier p' so that $p' \neq p$

Output: The well-formed iteration activity:

$$\text{iteration } p' \text{ do } \mathbf{A}$$

5.1.3 Deletion Change Operators

Deletion change operators allow the removal of activities nested within other activities or the root choreography itself.

5.1.3.1 Remove Activity from Block

This change operator allows to remove an activity located at a specified position in an already-existing block. All the activities nested in the activity that is removed (if this last one is a complex activity, see Section 3.2), are removed “on-cascade.”

Input 1: A well-formed block:

$$\{\mathbf{A}_1; \dots; \mathbf{A}_n\}$$

Input 2: An index i so that: $1 \leq i \leq n$

Precondition (Syntactic Correctness): Blocks must contain at least one nested activity. To prevent the output block from violating the syntax of ChorTex, the input block must contain at least two nested activities, namely:

$$n \geq 2$$

Output: The well-formed block:

$$\{\mathbf{A}_1; \dots; \mathbf{A}_{i-1}; \mathbf{A}_{i+1}; \dots; \mathbf{A}_n\}$$

5.1.3.2 Remove Branch from Choice Activity

This change operator allows to remove a branch from an already-existing choice activity. All the activities nested in the block that is removed are deleted “on-cascade.”

Input 1: A well-formed choice activity:

$$\text{choice } p \text{ either } \mathbf{A}_{b_1} \text{ or } \dots \text{ or } \mathbf{A}_{b_n}$$

Input 2: An index i so that: $1 \leq i \leq n$

Precondition (Syntactic Correctness): Choice activities must specify at least two branches.

To prevent the output choice activity from violating the syntax of ChorTex, the input choice activity must specify at least three branches:

$$n \geq 3$$

Output: The well-formed choice activity:

$$\text{choice } p \text{ either } \mathbf{A}_{b'_1} \text{ or } \dots \text{ or } \mathbf{A}_{b'_{n-1}}$$

The branches of the output choice activity is equals to the set of the branches of the input choice activity, minus \mathbf{A}_{b_i} :

$$\{\mathbf{A}_{b'_1}, \dots, \mathbf{A}_{b'_{n-1}}\} := \{\mathbf{A}_{b_1}, \dots, \mathbf{A}_{b_n}\} \setminus \{\mathbf{A}_{b_i}\}$$

5.1.3.3 Remove Branch from Parallel Activity

This change operator allows to remove a branch from an already-existing parallel activity. All the activities nested in the block that is removed are also removed “on-cascade” from the choreography.

Input 1: A well-formed parallel activity:

$$\text{parallel do } \mathbf{A}_{b_1} \text{ and } \dots \text{ and } \mathbf{A}_{b_n}$$

Input 2: An index i so that: $1 \leq i \leq n$

Precondition (Syntactic Correctness): Parallel activities must specify at least two branches.

To prevent the output parallel activity from violating the syntax of ChorTex, the input parallel activity must specify at least three branches:

$$n \geq 3$$

Output: The well-formed parallel activity:

$$\text{parallel do } \mathbf{A}_{b'_1} \text{ and } \dots \text{ and } \mathbf{A}_{b'_{n-1}}$$

The branches of the output parallel activity is equals to the set of the branches of the input parallel activity, minus \mathbf{A}_{b_i} :

$$\{\mathbf{A}_{b'_1}, \dots, \mathbf{A}_{b'_{n-1}}\} := \{\mathbf{A}_{b_1}, \dots, \mathbf{A}_{b_n}\} \setminus \{\mathbf{A}_{b_i}\}$$

5.1.3.4 Remove Named Exception Handler from Choreography

This change operator allows the removal of a named exception handler from an existing choreography. The body of the removed named exception handler, as well as the activities nested in it, are deleted “on-cascade.”

Input 1: A well-formed choreography with at least one named exception handler:

$$\text{chor } (\mathbf{A} \mid e_1 : \mathbf{A}_{e_1} \mid \dots \mid e_n : \mathbf{A}_{e_n} \mid * : \mathbf{A}_*)$$

Input 2: An exception type e so that: $e \in \{e_1, \dots, e_n\}$

Output: The well-formed choreography:

$$\text{chor } (\mathbf{A} \mid e'_1 : \mathbf{A}_{e_1} \mid \dots \mid e'_{n-1} : \mathbf{A}_{e_{n-1}} \mid * : \mathbf{A}_*)$$

The set of exception types that have associated a named exception handler in the output choreography is the set of exception types that have associated a named exception handler in the input choreography, minus e :

$$\{e'_1, \dots, e'_{n-1}\} := \{e_1, \dots, e_n\} \setminus \{e\}$$

For the sake of understandability, the input and output choreographies above specify a default exception handlers. This is meant only for simplifying the formal specification. The present change operator is obviously also applicable to choreographies that do not specify default exception handlers.

5.1.3.5 Remove Default Exception Handler from Choreography

This change operator allows the removal of the default exception handler specified by an existing choreography. The body of the removed default exception handler, as well as the activities nested in it, are deleted “on-cascade.”

Input: A well-formed choreography that specifies a default exception handler:

$$\text{chor } (\mathbf{A} \mid e_1 : \mathbf{A}_{e_1} \mid \dots \mid e_n : \mathbf{A}_{e_n} \mid * : \mathbf{A}_*)$$

Output: The well-formed choreography:

$$\text{chor } (\mathbf{A} \mid e_1 : \mathbf{A}_{e_1} \mid \dots \mid e_n : \mathbf{A}_{e_n})$$

Notice that, for the sake of understandability, the input and output choreographies above specify named exception handlers; however, this change operator is obviously also applicable to choreographies that do not specify named exception handlers.

5.1.4 Combining Change Operators

The change algebra hereby presented is *minimal*, meaning that it is defined as the smallest set of change operators which, combined with each other, allow the modification of every aspect of a well-formed choreography. For example, consider the case one wants to remove one of the two branches of a parallel activity using the “Remove Branch from Parallel Activity” change operator. Just applying the “Remove Branch from Parallel Activity” would not work, because the resulting parallel activity would have only one branch and, therefore, would not be syntactically valid. Instead, one can do one of the two following things, which yield choreographies that are equivalent in terms of operational semantics, but structurally different:

- Add to the parallel activity a “semantically void” branch (a.k.a. a “dummy branch”) containing just a skip activity nested in a block, and then remove the other undesired branch;
- Delete the entire parallel activity and then add again only the one branch that should have stayed in the parallel activity.

For the sake of brevity, such sequences of change operators for accomplishing complex modifications will be discussed only when they are necessary, e.g., in the scope of remediation strategies.

5.2 A Framework of Remediation Strategies and Plans

The ultimate goal of this thesis is to provide choreography modelers with automatically calculated remediation plans that can fix the realizability defects affecting their ChorTex choreographies. The algorithms for calculating remediation plans are called remediation strategies. The aim of the present section is to lay down the foundation of the correction process for unrealizable ChorTex choreographies by discussing the desiderata for remediation plans (Section 5.2.1) and the general structure of remediation strategies (Section 5.2.2).

5.2.1 Desiderata for Remediation Plans

A remediation plan is a change that, applied to a choreography affected by one or more realizability defects, solves at least one of the latter. By this definition, we can elicit the first requirement for remediation plans:

Remediation Principle (Usefulness of remediation plans). A remediation plan *must* solve at least one of the realizability defects afflicting the choreography for which it has been calculated.

Notice that it might be the case that a remediation plan targeted at solving one particular realizability defect might actually have, as side effect, the correction of other realizability defects as well.

As discussed in Section 1.2, the process of correcting unrealizable choreographies by means of remediation plans is iterative in nature. Ideally, the calculation of remediation plans is performed while the choreography is being modeled. The choreography may be affected by any number of realizability defects and, for each of them, one or more alternative remediation plans are proposed to the modeler. The modeler may elect to apply one of the proposed remediation plans, hence solving at least one realizability defect. By applying the remediation plan, the choreography is modified and, if there still are realizability defects affecting it, new remediation plans are immediately calculated.

In order for the correction process to be helpful to choreography modelers, however, it must be *finite*: given a choreography with a given number of realizability defects, there must be a finite amount of iterations necessary to reach a defects-free choreography. The iterative nature of the correction process naturally lends itself to the application of mathematical induction: if, after every application of a remediation plan, there are strictly fewer realizability defects than before, then the entire choreography can be fixed in a finite amount of iterations. Moreover, the amount of iterations necessary to fix the realizability of a choreography are fewer or equal in number to the amount of defects in the original choreography. Applying at most as many remediation plans as realizability defects in the original choreography is definitely finite² as well as reasonable from a practical perspective. The following requirement is fundamental in achieving the goal of correcting the n realizability defects of a choreography by applying at most n remediation plans:

Remediation Principle (Primum non nocere). The application of a remediation plan to the choreography for which it has been calculated *must not* introduce new realizability defects.

²Even assuming that every node in the Awareness Model is affected by all possible realizability defects for each of the participants involved in the choreography, this still yields a finite, however large upper-bound to the amount of realizability defects in a given choreography.

The principle of “Primum non nocere” (Latin for “first, do no harm”) comes from the medical domain³ and it captures the fundamental directive of not harming a patient with a treatment. Imagine the case in which the correction of a realizability defect by means of a remediation plan may introduce further realizability defects. Due to the iterative nature of the correction process, modelers may be caught in a “whirlpool” of remediation plans that, by introducing every time new realizability defects, prevent the choreography from ever becoming realizable. Combined, the “Usefulness of remediation plans” and “Primum non nocere” principles guarantee that, assuming each possible realizability defect can be corrected in a way that is acceptable to the modeler, all realizability defects in a choreography can be solved in a finite amount of iterations of the correction process.

5.2.2 Outlook of Remediation Strategies

In order to produce remediation plans that satisfy the “Usefulness of remediation plans” principle, each of the remediation strategies presented in the remainder of this chapter focuses on one of the types of realizability defects that have been discussed in Section 4.5.1. In particular, remediation strategies are proposed for each of the three types of realizability defects are presented in Section 5.4.1 through Section 5.4.3.

In order to clarify the formulation of the remediation strategies, it is useful to briefly describe the process that lead to them. Due to the goal of adhering to the “Primum non nocere” principle, the initial work on defining remediation strategies has been a laborious, iterative process. The first version of the remediation strategies strived to incorporate both the “Usefulness of remediation plans” and “Primum non nocere” principles. However, “hard-coding” from the very beginning the “Primum non nocere” principle in the remediation strategies has proven an overwhelmingly hard endeavor. The complexity of each remediation strategy grew quickly out of control, mostly because each had to fundamentally have “built-in” an almost complete realizability analysis for the choreography *resulting from the application of the remediation plans that could be produced*. Moreover, the prototyping of the first few remediation strategies devised in this first attempt has proved to be very prone to bugs and unforeseen “corner-cases,” not to mention that the remediation strategies themselves were extremely hard to be effectively documented in this chapter.

So, a paradigm shift was in order. Ensuring the “Primum non nocere” principle was an incredibly daunting task because the many change operators defined in the ChorTex change algebra could combine with each other in simply too many ways to foresee and handle. Moreover, not all change operators are needed to define a rich, versatile library of remediation strategies. Indeed, as discussed in Section 5.3, only a relatively narrow subset of the change algebra is actually required. After having identified a subset of the change algebra that seemed adequate to define sufficient remediation strategies, the next step was to investigate how the application of each of these change operators may result in the introduction of new realizability defects. This research has lead to the definition of *invariants*, presented in Section 5.3, that prevent the introduction of new realizability defects using the selected change operators. The invariants embody the “Primum non nocere” principle and are used in the definition of the remediation strategies presented in Section 5.4 as “filters” for remediation plans that would violate the “Primum non nocere” principle. With the “Primum non nocere” principle thus taken care of, the reminder of the formulation of remediation strategies consisted in upholding the “Usefulness of remediation plans” principle, which has proven to be a very straightforward task.

Notice, however, that invariants could not be formulated for every possible application of change operators of interest to the remediation plans. Sometimes, the complexity of foreseeing all the possible ways in which a change would affect Awareness Models and the respective participant awareness is simply too big. In such cases, we resort to *empirical verification* of remediation plans, i.e., applying them “in the background” to a copy of choreography, analyze the realizability of the resulting, modified choreography, and check that the latter has no new realizability defects when compared to the original choreography. If the modified choreography has no new realizability

³Where it is also known as the “non-maleficence” principle.

defects with respect to the original one, the remediation plan is proposed to the modeler; otherwise, the remediation plan is discarded. This “trial and error” approach may seem wanting for elegance, but it is actually very pragmatic and, in our experience with the prototype, seems to not impact the responsiveness and effectiveness of the correction process.

5.3 On Introducing new Realizability Defects

The “Primum non nocere” remediation principle states that the correction of realizability defects must not have as side-effect the introduction of new realizability defects. Therefore, the remediation strategies and plans that are the ultimate goal of this chapter must be built on top of an understanding of *how not to modify* a ChorTex choreography in order not to compromise its realizability (possibly further, if it already has some realizability defects). The goal of this section is to correlate the change operations defined in Section 5.1 with the causes of the three types of realizability defects for ChorTex choreographies identified in Section 4.5.1.

In the remainder of this section, the three different types of realizability defects are treated separately: each one is discussed in terms of how the three different types of change operators, namely insertion, update and deletion, can be (mis)used to introduce new instances of that realizability defect type. As already done for the realizability analysis method presented in Chapter 4, we assume the choreography that is undergoing modification to be well-formed (see Section 3.5).

The discussion of invariants for change operators is restricted to the subset of the change algebra listed in Table 5.1, which relates change operators to the respective invariants. It should be noted that the “scope of applicability” of the “Insert Activity in a Block” and “Remove Activity from Block” change operators will be restricted to inserting and removing activities of one of the following four types: message exchange activities, opaque activities, choice activities and iteration activities. This restriction makes sense in the scope of this work because no remediation strategies employ “Insert Activity in a Block” and “Remove Activity from Block” change operators differently. Additionally, we assume that instances of the activities that are inserted or deleted do not have exceptions propagating from them. (The reason for this last assumption is laid out in a few paragraphs when discussing “disruptive change operators.”)

As shown in Figure 5.2, the change operators that we are not discussing in the remainder of this section fall in one of the following two categories: those that are *irrelevant* towards introducing new realizability defects and those that are *too disruptive* in terms of changes to the Awareness Model underpinning the choreography.

Irrelevant change operators: “Change Name of Activity” and “Change Message Type of Message Exchange Activity” are the two change operators that cannot introduce new realizability defects. In fact, neither activity names nor message type identifiers have any effect on the structure of the distributed messaging behavior specified by the choreography.

Activity names are just unique identifiers of activities and, while they do appear in the enactment traces inside the actions specified by those activities, they have no real import in terms of operational semantics.

The argument we made for activity names applies to message type identifiers as well, but only because we are operating under the assumption that no two message exchange activities can use the same message type in a well-formed choreography due to the “Unique Message Types” well-formedness requirement. If the “Unique Message Types” well-formedness requirement were to be removed or somewhat relaxed, applications of the “Change Message Type of Message Exchange Activity” change operator may lead to “message type clash,” i.e., two or more message exchange activities that consist of the same messages of the same type being dispatched. As already discussed in Section 3.5, relaxing the “Unique Message Types” well-formedness requirement may lead to “non-deterministic choice” realizability defects (see Section 2.3.4.4), namely when the recipients of a message misunderstand which message exchange activity has generated it and therefore they

Change Operator	Type 1	Type 2	Type 3
Insert Activity in a Block	<ul style="list-style-type: none"> • Invariant 1 • Invariant 2 • Invariant 3 • Invariant 4 	<ul style="list-style-type: none"> • Invariant 9 • Invariant 10 	<ul style="list-style-type: none"> • Invariant 15 • Invariant 16
Change Sender of Message Exchange Activity	<ul style="list-style-type: none"> • Invariant 5 • Invariant 6 	<ul style="list-style-type: none"> • Invariant 11 • Invariant 12 	<ul style="list-style-type: none"> • Invariant 17 • Invariant 18
Add Recipient to Message Exchange Activity	<i>none</i>	<i>none</i>	<i>none</i>
Remove Recipient from Message Exchange Activity	<ul style="list-style-type: none"> • Invariant 6 	<ul style="list-style-type: none"> • Invariant 12 	<i>none</i>
Add Participant to Opaque Activity	<ul style="list-style-type: none"> • Invariant 5 	<ul style="list-style-type: none"> • Invariant 11 	<ul style="list-style-type: none"> • Invariant 18
Remove Participant from Opaque Activity	<ul style="list-style-type: none"> • Invariant 6 	<ul style="list-style-type: none"> • Invariant 12 	<i>none</i>
Change Decision Maker of Choice Activity	<ul style="list-style-type: none"> • Invariant 5 • Invariant 6 	<ul style="list-style-type: none"> • Invariant 11 • Invariant 12 	<ul style="list-style-type: none"> • Invariant 17 • Invariant 18
Change Decision Maker of Iteration Activity	<ul style="list-style-type: none"> • Invariant 5 • Invariant 6 	<ul style="list-style-type: none"> • Invariant 11 • Invariant 12 	<ul style="list-style-type: none"> • Invariant 17 • Invariant 18
Remove Activity from Block	<ul style="list-style-type: none"> • Invariant 7 • Invariant 8 	<ul style="list-style-type: none"> • Invariant 13 • Invariant 14 	<ul style="list-style-type: none"> • Invariant 19

Table 5.1: Change operators and the invariants they have to respect not to introduce new realizability defects.

come to wrong assumptions on the current state of the enactment and the behavior they expect will follow.

Disruptive change operators: These change operators have potentially very strong influence on the structure of the Awareness Models underpinning the modified choreographies. Participant-awareness, which is the foundation of the realizability analysis method presented in Chapter 4, is built on top of the structure of the Awareness Model and, more specifically, on the actions represented by the nodes and the network of control-edges connecting them. The goal of this section is to study how (not) to change a choreography in order (not) to introduce realizability defects. The ultimate goal is to derive invariants that guarantee that a given change does not introduce new realizability defects. These invariants are necessarily defined on the choreography and its Awareness Model as they are *before* the application of a change. Their logic, however, is aimed at constraining how the participant awareness can be *after* the change has been applied, i.e., that the change does not cause new realizability defects.

The change operators that we labelled as “disruptive” affect the structure of the Awareness Model in such a way that we cannot capture with invariants. Consider, for example, a change that introduces a new throw activity in a block. Due to the mapping from ChorTex activities to Awareness Models sub-graphs presented in Section 3.4.1, it is very likely the case that this change would make reachable some parts of the Awareness Model that were previously unreachable from the start node (and thus so far ignored by the realizability analysis method) and vice-versa. These changes in the structure of the Awareness Model are not “localized,” meaning that they very often affect parts of the Awareness Model that are not generated from the activities been modified by the change or those close to them. The variability of the Awareness Models because of disruptive changes is such that we do not know how relate the participant awareness *before* the change to the one *after* the change in a way generic enough be conducive to the definition of the invariants. (It should be noted, however, that if one were in need to know if a disruptive change would introduce new realizability defects, there is always the option of empirical verification.)

5.3.1 How to Introduce new Type 1 Realizability Defects

Type 1 realizability defects (“Unawareness of non-initiating action enactability”) occur when one acting participant cannot observe its action becoming enactable (see Section 4.4.1). The key for remediating Type 1 realizability defects lies in the Awareness Annotation Algorithm, namely the algorithm used for annotating the participant awareness on an Awareness Model (see Section 4.3). Recall that the participant awareness is “split” in two aspects with respect to actions: awareness that the action becomes enactable and the action being enacted (see Section 4.2). The awareness of the participant p with respect of the action a becoming enactable is denoted as $AW_{\text{fireable}}(p, n_a)$, where n_a is the node in the Awareness Model representing the enactment of a . A realizability defect of Type 1 occurs for the acting participant p at the node n_a when that participant is neither immediately nor eventually aware of n_a becoming enactable, which is formally denoted by the predicate (see Section 4.4.1.1):

$$p \in \text{actingParticipants}(n_a) : AW_{\text{fireable}}(p, n_a) \notin \{\text{ia}, \text{ea}\}$$

Given the fact that the other only two alternative values for participant awareness are unawareness and non-involvement, the following predicate is equivalent to the previous:

$$p \in \text{actingParticipants}(n_a) : AW_{\text{fireable}}(p, n_a) \in \{\text{ua}, \text{mi}\}$$

Put it in natural language, a Type 1 realizability defect occurs for an acting participant p of the node n_a if p is either unaware or non-involved of n_a becoming enactable.

Insufficient participant-awareness of a node becoming enactable is the cause of Type 1 realizability defects. And the participant-awareness of a node becoming enactable is the result of the participant-awareness resulting from the enactment of that node’s predecessors in the Awareness Model. The Awareness Annotation Algorithm is a flow algorithm in which the edges of the

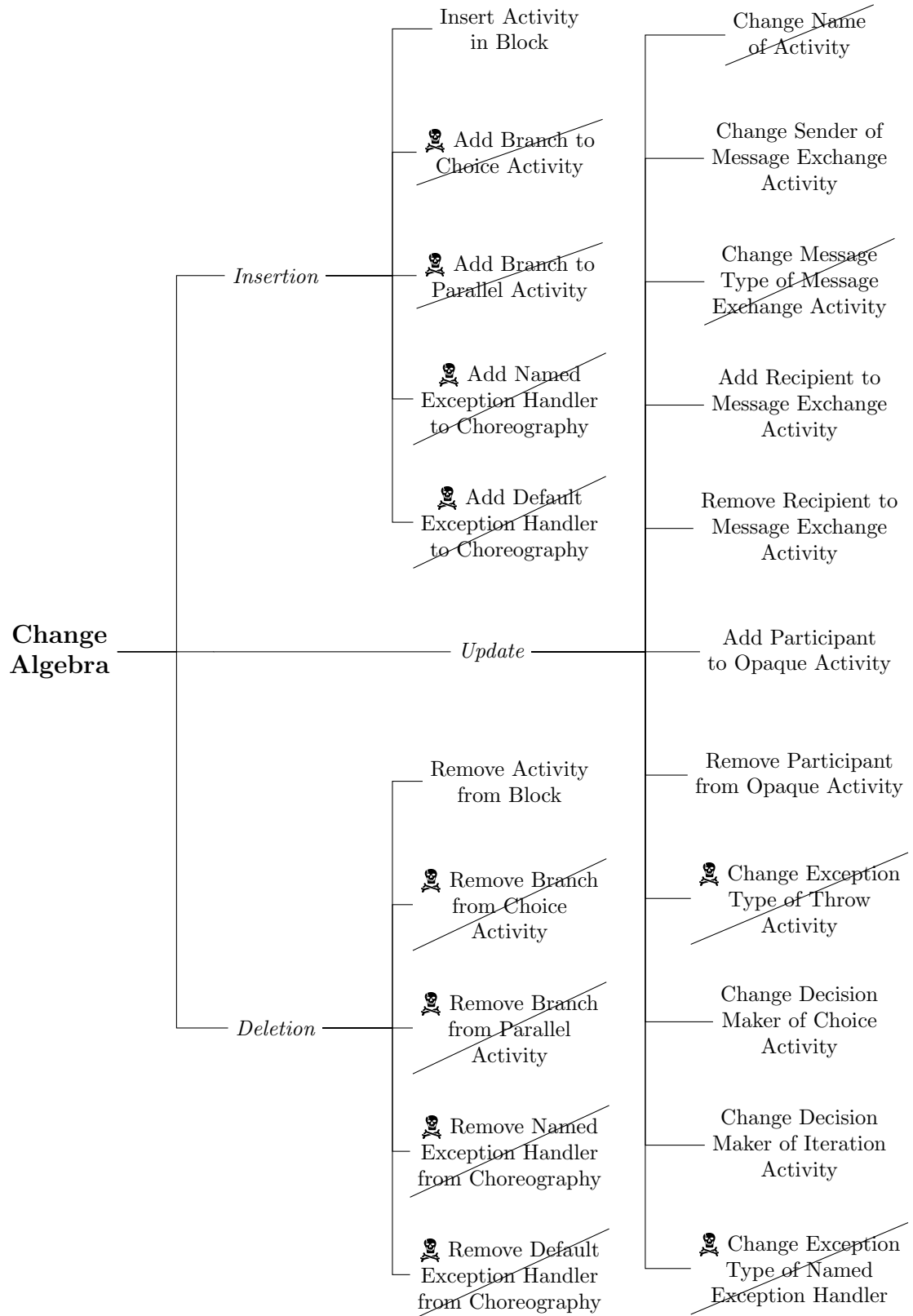


Figure 5.2: Change operators considered for the invariants and those not considered. The change operators not considered are striked out. Those not considered because too disruptive are marked with the symbol, the others are irrelevant.

Awareness Model are traversed from source to target, that is, following the direction of the arrows. Therefore, $AW_{\text{fireable}}(p, n_a)$ is the combination of the participant awareness of p resulting from the execution of other actions by either p or by some other participants. More specifically, $AW_{\text{fireable}}(p, n_a)$ is based on the result of the merge, calculated by means of the safe-approximation function $\hat{\text{A}}$ defined in Section 4.3.2, of the AW_{fired} values for p for the predecessor nodes of n_a . Put formally:

$$AW_{\text{fireable}}(p, n_a) \approx \hat{\text{A}}(AW_{\text{fired}}(p, n_{a_1}), \dots, AW_{\text{fired}}(p, n_{a_m}))$$

It should be noted that in the equation above we use the symbol \approx to denote that, actually, the safe-approximation of the AW_{fired} values of n_{a_1}, \dots, n_{a_m} may not be exactly the value of $AW_{\text{fireable}}(n_a)$. This is the case, however, if and only if there is at least one path connecting one of the nodes n_{a_1}, \dots, n_{a_m} with n_a that traverses a parallel activity. The difference between $AW_{\text{fireable}}(n_a)$ and the safe-approximation of $AW_{\text{fireable}}(n_{a_1}), \dots, AW_{\text{fireable}}(n_{a_m})$ may consist in one or more participants being non-involved in the former and unaware in the latter (see Section 4.3.2). Type 1 realizability defects occur whether an acting participant is either non-involved or unaware, and therefore this difference is irrelevant to the end of defining remediation strategies for Type 1 realizability defects.

Recall that traversing reactive nodes (i.e., nodes not representing actions, see Definition 3.8) does not modify participant awareness (see Section 4.2). Therefore, the value of $AW_{\text{fireable}}(p, n_a)$ is the result of the safe-approximation of values of $AW_{\text{fired}}(p)$ of those participant-activated nodes from which n_a is reachable by traversing only reactive nodes. This relationship among participant-activated nodes in the AWM is captured by the notion of *awareness dependency*.

Definition 5.2 (Awareness Dependency). A node n in the Awareness Model is *awareness-dependent* on the participant-activated node n' , which is said to be an *awareness-dependee* of n , if there is a path in the Awareness Model connecting n' and n made exclusively of reactive nodes. The notations $\mathbb{D} \downarrow (n)$ and $\mathbb{D} \uparrow (n)$ denote the set of awareness-dependee and awareness-dependent nodes of the node n , respectively.

It is important to notice that the above definition of awareness dependency implies that not only participant-activated nodes have awareness dependencies: reactive nodes have awareness dependencies too. However, all awareness-dependee nodes, i.e., those nodes that are “targeted” by awareness dependencies, are necessarily participant-activated nodes. Figure 5.3 presents on the running example on Chapter 4 the awareness dependencies of the nodes that have associated awareness constraints (representing all awareness dependencies would have resulted in an Awareness Model too cluttered to read). For example, consider the node derived from the message exchange activity mex_3 , which represents the dispatching of a message of type m_3 by p_3 to p_1 and p_2 . This node has two awareness dependees, namely the nodes mex_1 and mex_2 , which represent the dispatching of messages of type m_1 and m_2 , as they are the only participant-activated nodes from which mex_3 can be reached without traversing other participant-activated nodes.

Awareness dependencies are the key to realizability defects of Type 1, as they allow to relate the insufficient awareness of an acting participant with respect to its action becoming fireable with the other actions that cause this insufficient awareness. Changes that modify existing or introduce new awareness dependencies may introduce new Type 1 realizability defects, and so can changes that affect how the traversal of a participant-activated node modifies the participant-awareness. In the remainder we use the expression *unsatisfactory awareness dependees* to refer to those nodes that, due to insufficient participant awareness, cause realizability defects of Type 1 in some of their awareness-dependent nodes. In the example shown in Figure 5.3, mex_2 is the only unsatisfactory awareness dependee of mex_3 .

5.3.1.1 Type 1 Realizability Defects and Insertion Change Operators

There are four types of activities that specify actions: message exchange activities, opaque activities, choice activities and iteration activities (see Definition 3.1). Given the syntax of ChorTex, instances of these four types can be nested exclusively into blocks activities (see Section 3.2). Message exchange and opaque activities are *atomic*, meaning that no other activity can be nested in

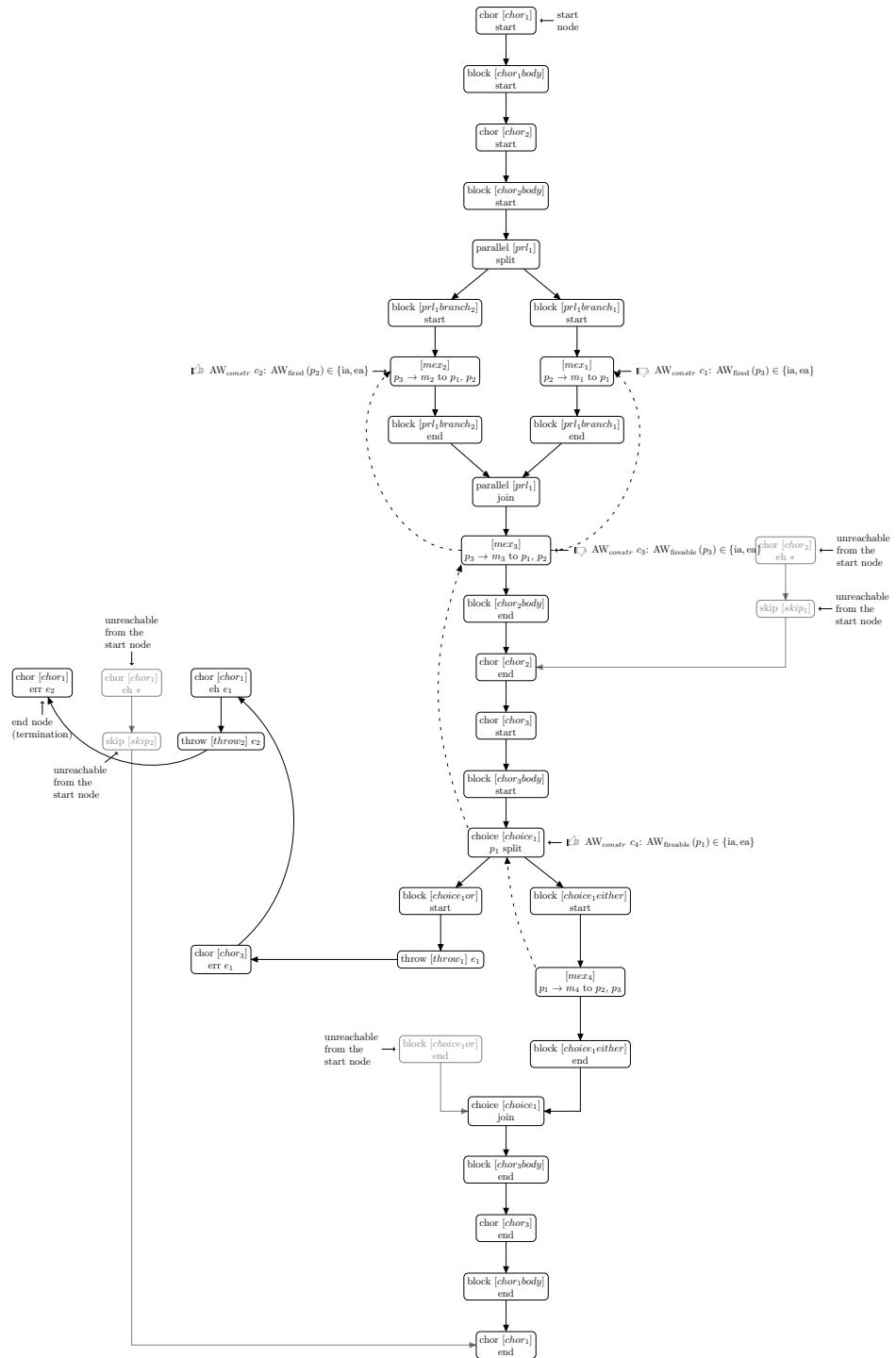


Figure 5.3: The Awareness Model presented in Figure 3.3, augmented with the awareness constraints and the awareness dependencies (depicted with dashed arrows connecting awareness-dependent nodes with their awareness depends) of the nodes with awareness constraints.

them. Choice and iteration activities, instead, are *composite* activities that *require* other activities to be nested into them in order for the surrounding choreography to be well-formed. In particular, these activities must be blocks, which are also composite activities. In the remainder, we treat atomic and composite activities separately.

Atomic activities: Atomic activities are mapped in the Awareness Models to single participant-activated nodes (see Section 3.4.1). When inserted into blocks, these participant-activated nodes are connected through sequence flows to the last node of the previous activity in the block (or the block’s start node, if the atomic activity is the first activity nested in the block) and the start node of the next activity in the block (or the end node of the block if the atomic activity is the last one in the block).

The insertion of participant-activated nodes in the Awareness Model may have the effect of modifying existing awareness dependencies. Consider the example shown in Figure 5.4. The two participant-activated nodes n_{d_1} and n_{d_2} are the only awareness dependees of a node n_b . Upon the insertion of n “between” the nodes n_a and n_b (where the suffixes b and a stand for “before” and “after,” respectively) the nodes n_{d_1} and n_{d_2} become awareness-dependee nodes of n , which in turn becomes the only awareness-dependee node of n_b .

Shift of awareness dependencies caused by insertion operators happen in many cases, but it is not the rule. Consider Figure 5.5: since the node mex_3 does not lay on all paths connecting mex_1 with mex_2 , mex_3 does not replace mex_1 as the only awareness dependee of mex_2 (i.e., there is no shift in awareness dependencies for mex_2); instead, mex_3 simply becomes another awareness dependee of mex_2 .

When inserting single participant-activated node between two other nodes, its resulting awareness dependees are straightforward to predict: they will be exactly those nodes that are currently awareness dependees for the second node. Instead, the nodes for which the new one will be an awareness dependee are all those that, before the insertion, are reachable from the first node without traversing participant-activated nodes.⁴

Let n be the new participant-activated node resulting from the insertion change; new Type 1 realizability defects can be introduced as result of the insertion of n :

- On the node n ;
- On the awareness-dependent nodes of n .

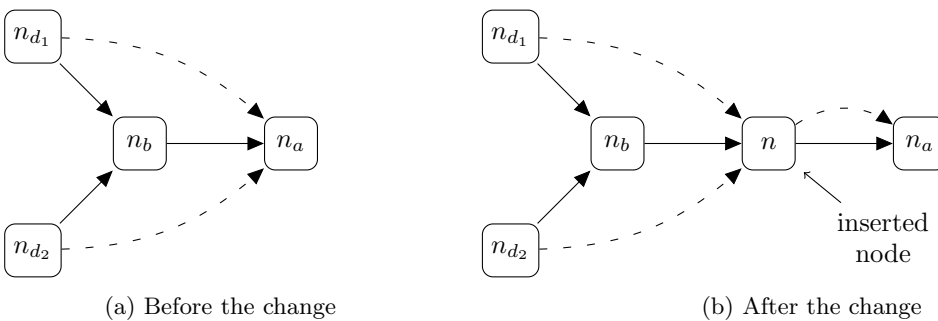


Figure 5.4: Example of the effect of the insertion of an atomic activity on an Awareness Model that causes a shift in awareness dependencies.

⁴In the implementation described in Chapter 6, we use a trick to “cache” this information: we have extended the notion of awareness dependency to “quasi-awareness dependency,” in which also reactive nodes can be awareness dependee of other nodes. All the algorithms and methods discussed in the remainder of this Chapter do not suffer from this relaxed notion of awareness dependency because reactive nodes do not modify the participant awareness, and therefore can be ignored in the scope of the \mathfrak{A} safe-approximation function.

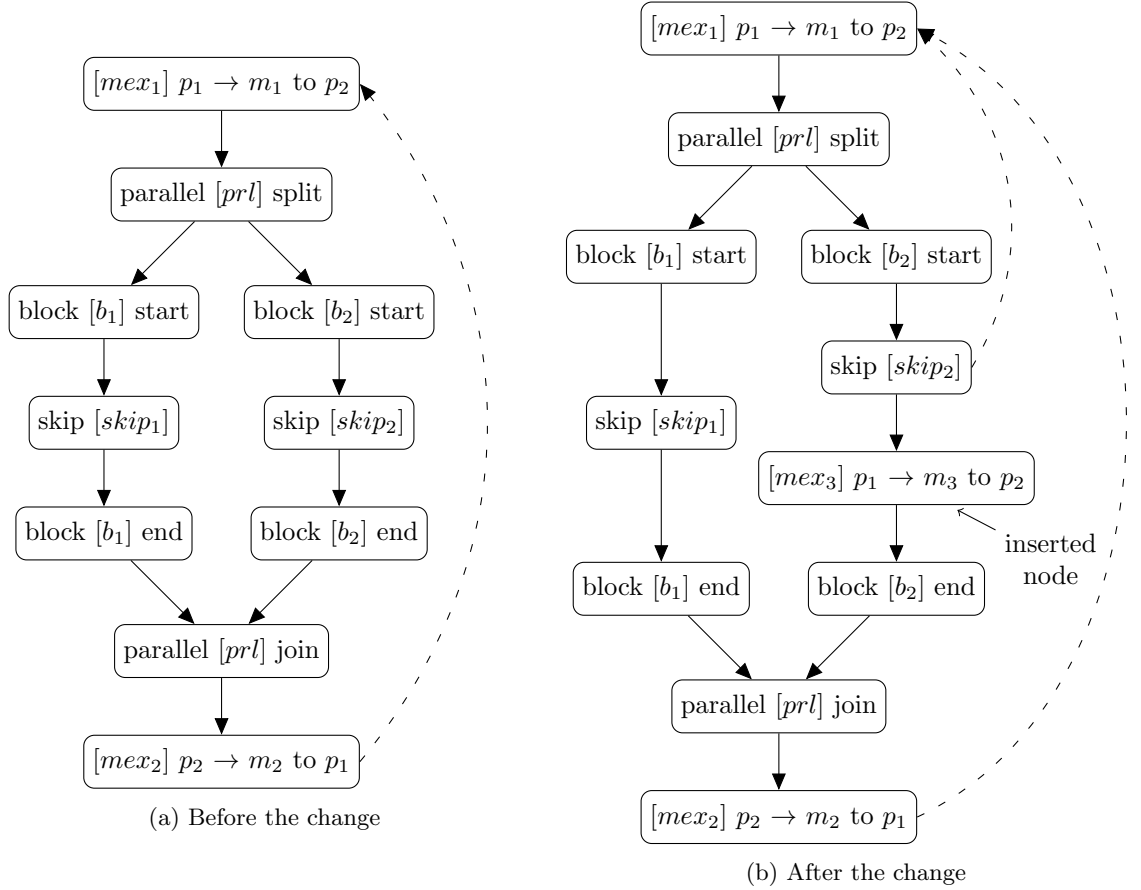


Figure 5.5: Example of the effect of the insertion of an atomic activity on an Awareness Model that does not cause a shift in awareness dependencies.

A new Type 1 realizability defect is introduced on the node n if, as a result of the change, its awareness dependees become unsatisfactory. Given the start node n_s before which n is inserted, we can formalize the introduction of such a new Type 1 realizability defect as follows:

$$\exists n_d \in \mathbb{D} \downarrow (n) \exists p \in \text{actingParticipants}(n) : \text{AW}_{\text{fired}}(n_d, p) \notin \{\text{ia}, \text{ea}\}$$

By negating the predicate above, we obtain the following invariant:

Invariant 1. The insertion in a block of an atomic activity mapped to the participant-activated node n before the activity that has n_s as start node does not result in a new Type 1 realizability defect on the newly-inserted participant activated node if and only if:

$$\forall n_d \in \mathbb{D} \downarrow (n_s) \forall p \in \text{actingParticipants}(n) : \text{AW}_{\text{fired}}(n_d, p) \in \{\text{ia}, \text{ea}\}$$

A new Type 1 realizability defect is introduced on an awareness-dependent node n_d of n if, after the change, n is an unsatisfactory awareness dependee for n_d . Given the node n_e after which n is inserted, we can formalize the introduction of such a new Type 1 realizability defect as follows:

$$\exists n_d \in \mathbb{D} \uparrow (n_e) \exists p \in \text{actingParticipants}(n_d) : \text{AW}_{\text{fired}}(n_d, p) \notin \{\text{ia}, \text{ea}\}$$

By negating the predicate above, we obtain the following invariant:

Invariant 2. The insertion in a block of an atomic activity mapped to the participant-activated node n after the activity that has n_e as end node does not result in a new Type 1 realizability defect on any awareness-dependent node of n if and only if:

$$\forall n_d \in \mathbb{D} \uparrow (n_e) \forall p \in \text{actingParticipants}(n_d) : \text{AW}_{\text{fired}}(n, p) \in \{\text{ia}, \text{ea}\}$$

Composite activities: We assume that none of the blocks nested in the choice or iteration activity to be inserted contains at any nesting level a throw activity (or that, if there are some, the exceptions they throw are handled within the inserted activity and do not propagate outside of it). Under this assumption, the sub-graphs constructed from such choice and iteration activities have exactly one start node and one end node, which coincide in the case of iteration activities (see Section 3.4.1).

Consider the example shown in Figure 5.6, which presents how a snippet of an Awareness Model is modified by the insertion of a choice activity with two branches, each branch containing one message exchange or opaque activity. (The Awareness Model “after” the change shown in Figure 5.6b is simplified, as there would usually be reactive nodes represent start or end of blocks between n_s and n_{p_1} , n_s and n_{p_2} , n_{p_1} and n_e as well as n_{p_2} and n_e ; since we are omitting only reactive nodes, however, this does not affect the change in terms of awareness dependencies.) The node n_s represents the choice split node, which the a participant-activated node representing the decision about which branch to enact. The nodes n_{p_1} and n_{p_2} are participant-activated nodes representing the activities in the branches. The inserted subgraph is “closed” by the node n_e , which constitutes the choice-join node.

The conditions for the introduction of Type 1 realizability defects upon inserting a composite activity are similar to those for atomic activities. Consider the composite activity a with n_s and n_e as start and end nodes, respectively. Similarly to the case of atomic activities, new Type 1 realizability defects can be introduced:

- On the node n_s ;
- On the awareness-dependent nodes of n_e .

A new Type 1 realizability defect is introduced on the start node n_s if, after the change, its awareness dependees are not satisfactory; put formally:

$$\exists n_d \in \mathbb{D} \downarrow (n_s) \exists p \in \text{actingParticipants}(n_s) : \text{AW}_{\text{fired}}(n_d, p) \notin \{\text{ia}, \text{ea}\}$$

Therefore, the following invariant guarantees that the insertion of a complex activity in a block does not result in a new Type 1 realizability defect on that activity’s start node.

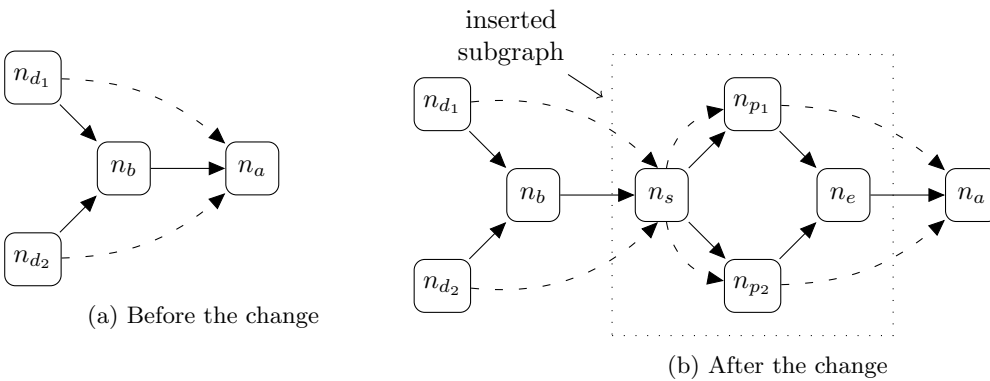


Figure 5.6: Example of the effect of the insertion of a choice activity on an Awareness Model.

Invariant 3. The insertion in a block of a complex activity with start node n_s before the activity that has n_a as start node does not result in a new Type 1 realizability defect on the newly-inserted participant activated node if and only if:

$$\forall n_d \in \mathbb{D} \downarrow (n_a) \forall p \in \text{actingParticipants}(n_s) : \text{AW}_{\text{fired}}(n_d, p) \in \{\text{ia}, \text{ea}\}$$

New Type 1 realizability defects can also be introduced on already-existing nodes, namely those that, due to the insertion change, become awareness-dependant of some of the new nodes inserted by the change. In this case there is a difference between choice and iteration activity. In the sub-graph of an iteration activity, the end node is always a participant-activated node, namely the one representing the decision of whether the activity is going to be iterated further or not. Since it is a participant-activated event, the end node of the iteration activity becomes awareness dependee of all the newly-introduced awareness dependencies. In the case of the choice activity, this is not the case, because the end node of a choice activity (the “join” node) is always a reactive event. The nodes in the sub-graph of the choice activity that will become awareness dependees of newly-introduced awareness dependencies are, however, the awareness dependees of the end node of the choice. Given the participant-activated nodes n_1, \dots, n_l in the sub-graph of the composite activity that become awareness dependees in newly-introduced awareness dependencies, new Type 1 realizability defects are introduced on their awareness-dependent nodes n'_1, \dots, n'_m if:

$$\exists i \in [1, l], j \in [1, m] \exists p \in \text{actingParticipants}(n'_j) : \text{AW}_{\text{fired}}(n_i, p) \notin \{\text{ia}, \text{ea}\}$$

The insertion change modifies the awareness-dependee nodes of all and only the nodes that, before the change, share awareness-dependees with the node “after” which the subgraph will be inserted (namely n_a in Figure 5.6b). The following invariant prevents the introduction of Type 1 on such nodes:

Invariant 4. The insertion in a block of a composite activity mapped with end node n_e after the activity that has n_b as end node and before the one that has n_a as start node does not result in new Type 1 realizability defect on any already-existing node if and only if:

$$\begin{aligned} \forall n_d \in \mathbb{D} \downarrow (n_e) \forall n'_d \in \mathbb{D} \downarrow (n_b) \forall n''_d \in \mathbb{D} \uparrow (n'_d) \\ \forall p \in \text{actingParticipants}(n''_d) : \text{AW}_{\text{fired}}(n_d, p) \in \{\text{ia}, \text{ea}\} \end{aligned}$$

5.3.1.2 Type 1 Realizability Defects and Update Change Operators

Update changes do not modify the structure of the Awareness Models underpinning the modified choreographies. Therefore, the awareness dependencies in an Awareness Model are not affected by change operators. However, the following update change operators, grouped by type of activity to which they can be applied, affect how the traversing of participant-activated nodes modifies the participant awareness (see Section 4.3.3), which can render existing awareness dependencies unsatisfied:

- Message exchange activity:
 - Change Sender of Message Exchange Activity
 - Add Recipient to Message Exchange Activity
 - Remove Recipient from Message Exchange Activity
- Opaque activity:
 - Add Participant to Opaque Activity
 - Remove Participant from Opaque Activity
- Choice activity:

- Change Decision Maker of Choice Activity
- Iteration activity:
 - Change Decision Maker of Iteration Activity

New Type 1 realizability defects can be introduced by the update change operators listed above both on the participant-activated nodes they affect, as well as the nodes that are awareness-dependent of them; we treat these two cases separately in the remainder.

New Type 1 realizability defects on updated nodes: The “Add Participant to Opaque Activity” change operator adds a new acting participant p to the affected participant-activated node n , while “Change Sender of Message Exchange Activity,” “Change Decision Maker of Choice Activity” and “Change Decision Maker of Iteration Activity” replace the previous acting participant with p . New Type 1 realizability defects are introduced on the updated node if its awareness dependees are not satisfactory for p , namely if:

$$\exists n_d \in \mathbb{D} \downarrow (n) : \text{AW}_{\text{fired}}(n_d, p) \notin \{\text{ia}, \text{ea}\}$$

Notice that removing an acting participant from a participant-activated node *cannot* introduce new Type 1 realizability defects on that node, because such a change does not result in the addition of new awareness constraints. Therefore, the “Remove Recipient from Message Exchange Activity” and “Remove Participant from Opaque Activity” cannot introduce new Type 1 realizability defects on the participant-activated nodes they affect.

Invariant 5. The application of an update change operator does not introduce a new Type 1 realizability defect on the affected non-initiating, participant-activated node n if and only if:

$$\forall n_d \in \mathbb{D} \downarrow (n) : \text{AW}_{\text{fired}}(n_d, p) \in \{\text{ia}, \text{ea}\}$$

New Type 1 realizability defects on awareness-dependent nodes: An update change to the participant-activated node n introduces new Type 1 realizability defects if, after the change, the AW_{fired} value of n does no longer satisfy the awareness-dependent nodes of n . When applying the “Remove Recipient from Message Exchange Activity” and “Remove Participant from Opaque Activity” removes the participant p from the affected participant-activated node n , its participant awareness in the AW_{fired} of n changes from eventually aware to unaware or not-involved. Similarly, the “Change Sender of Message Exchange Activity,” “Change Decision Maker of Choice Activity” and “Change Decision Maker of Iteration Activity” change the participant awareness of the previous acting participant p in the AW_{fired} of n from immediately aware to either unaware or not-involved. This results in new Type 1 realizability defects to all the awareness-dependent nodes of n of which p is an acting participant; put formally:

$$\exists n_d \in \mathbb{D} \uparrow (n) : p \in \text{actingParticipants}(n_d)$$

The “Add Recipient to Message Exchange Activity” and “Add Participant to Opaque Activity” change operators *increase* the participant awareness for the affected participant-activated node n (making the added participant in the AW_{fired} eventually aware instead of non-involved or unaware), therefore they cannot render unsatisfied awareness dependencies in which n is the awareness dependee.

Invariant 6. The application of an update change operator that removes p from the participants that can observe the traversal of the affected node n does not introduce a new Type 1 realizability defect on any of the awareness-dependent nodes of n if and only if:

$$\forall n_d \in \mathbb{D} \uparrow (n) : p \notin \text{actingParticipants}(n_d)$$

5.3.1.3 Type 1 Realizability Defects and Deletion Change Operators

Similarly to the case of insertion change operators, the removal of one participant-activated node or of a subgraph from an Awareness Model affects the awareness dependencies. We treat the removal of atomic and composite activities separately.

Atomic activities: The atomic activities that generate participant-activated nodes are message exchange and opaque activities. The removal of one such activity using the “Remove Activity from Block” causes a shift of awareness dependencies as exemplified in Figure 5.7. In a nutshell, the removal of the participant-activated node n introduces new awareness dependencies between the awareness-dependee and awareness-dependent nodes of n . New Type 1 realizability defects are introduced by the removal of an atomic activity when the awareness dependees of these new awareness dependencies are unsatisfactory to their awareness dependents; put formally:

$$\begin{aligned} \exists n_d \in \mathbb{D} \downarrow (n) \exists n'_d \in \mathbb{D} \uparrow (n) \exists p \in \text{actingParticipants}(n'_d) : \\ \text{AW}_{\text{fired}}(n_d, p) \notin \{\text{ia}, \text{ea}\} \end{aligned}$$

By negating the predicate above, we obtain the following invariant:

Invariant 7. The removal of an atomic activity that is mapped to the participant-activated node n does not introduce a new Type 1 realizability defect on any of the awareness-dependent nodes of n if and only if:

$$\begin{aligned} \forall n_d \in \mathbb{D} \downarrow (n) \forall n'_d \in \mathbb{D} \uparrow (n) \forall p \in \text{actingParticipants}(n'_d) : \\ \text{AW}_{\text{fired}}(n_d, p) \notin \{\text{ia}, \text{ea}\} \end{aligned}$$

Composite activities: As in the case of insertion change operators (see Section 5.3.1.1), we assume in the remainder that none of the blocks nested in a choice or iteration activity contains at any nesting level a throw activity (or that, if there are some, the exceptions they throw are handled within the activity and do not propagate outside of it). Under this assumption, the sub-graphs constructed from such choice and iteration activities have exactly one start node and one end node, which coincide in the case of iteration activities (see Section 3.4.1). Figure 5.8 exemplifies how a snippet of an Awareness Model is modified by the deletion of a choice activity with two branches, each of them containing one message exchange or opaque activity. Similarly to the case of the deletion of atomic activities, the deletion of a composite activity results in the introduction of new awareness dependencies between the awareness-dependee and awareness-dependent nodes of the start node of the composite activity. This, in turn, may result in new Type 1 realizability defects if the awareness-dependee nodes in these new awareness dependencies are unsatisfactory for their awareness-dependent nodes. Given the start node n_s of the deleted composite activity, we can

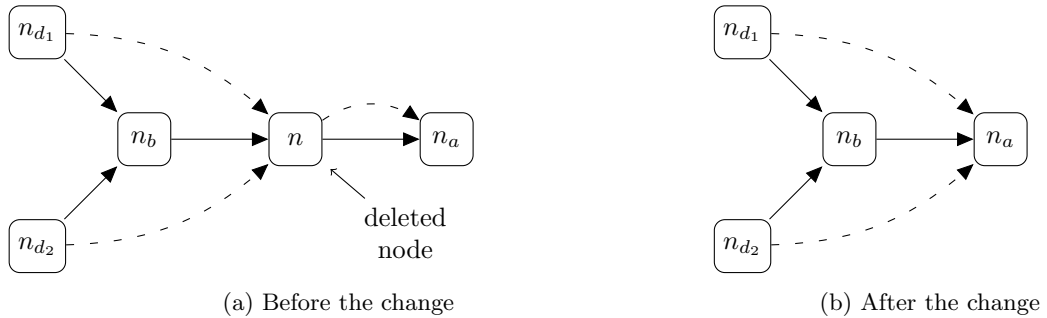


Figure 5.7: Example of the effect of the deletion of an atomic activity on an Awareness Model.

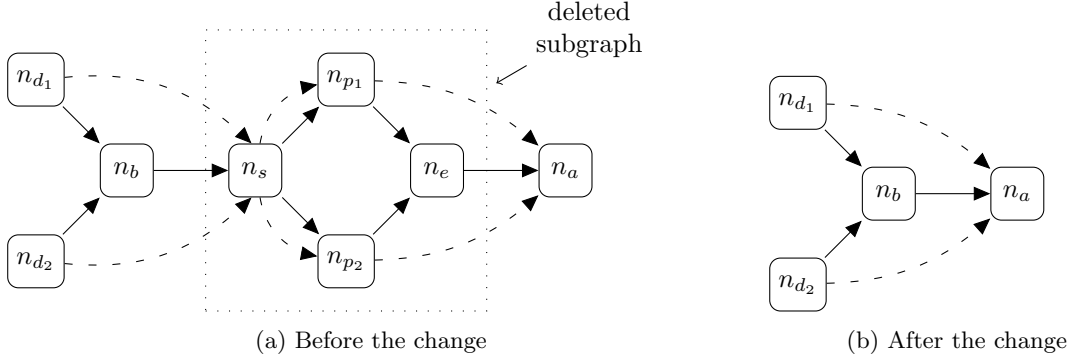


Figure 5.8: Example of the effect of the deletion of a choice activity from an Awareness Model.

formally characterize the introduction of new Type 1 realizability defects as follows:

$$\begin{aligned} \exists n_d \in \mathbb{D} \downarrow (n_s) \exists n'_d \in \mathbb{D} \uparrow (n_s) \exists p \in \text{actingParticipants}(n'_d) : \\ \text{AW}_{\text{fired}}(n_d, p) \notin \{\text{ia}, \text{ea}\} \end{aligned}$$

By negating the predicate above, we obtain the following invariant:

Invariant 8. The removal of a composite activity with start node n_s does not introduce a new Type 1 realizability defect on any of the awareness-dependent nodes of any of the deleted nodes if and only if:

$$\begin{aligned} \forall n_d \in \mathbb{D} \downarrow (n_s) \forall n'_d \in \mathbb{D} \uparrow (n_s) \forall p \in \text{actingParticipants}(n'_d) : \\ \text{AW}_{\text{fired}}(n_d, p) \notin \{\text{ia}, \text{ea}\} \end{aligned}$$

5.3.2 How to Introduce new Type 2 Realizability Defects

In a choreography with multiple initiating nodes, each of them is associated with an awareness constraint requiring that every acting participant of the other initiating actions is either immediately or eventually aware of the performing of the action represented by the node (see Section 4.4.1.2). Put formally, given the initiating nodes n_{a_1}, \dots, n_{a_k} representing them, each node $n_{a_i \in [1, k]}$, each is associated with the following awareness constraint for a certain participant p :

$$p \in \left(\bigcup_{j \in [1, \dots, i-1, i+1, \dots, k]} \text{actingParticipants}(a_j) \right) : \text{AW}_{\text{fireable}}(n_{a_i}, p) \in \{\text{ia}, \text{ea}\}$$

That is, a change to a choreography may introduce Type 2 realizability defects when it modifies the initiating actions and their acting participants, which can be done by means of insertion change operators (see Section 5.3.2.1), update change operators (see Section 5.3.2.2) and deletion change operators (see Section 5.3.2.3).

5.3.2.1 Type 2 Realizability Defects and Insertion Change Operators

The application of an insertion change operator to the choreography may result in the introduction of a new initiating action and, possibly, to actions previously initiating becoming non-initiating. The case of actions that change from initiating to non-initiating is covered by the discussion in Section 5.3.1.1, where the new initiating node is the newly-inserted participant-activated node.

When introducing a new initiating node, for example inside a new branch of a parallel activity, new Type 2 realizability defects can be introduced on the new initiating node as well as the other, pre-existing initiating nodes. When a message exchange or an opaque activity is inserted, the new

initiating node is, of course, the only participant-activated node to which the activity is mapped. When inserting a choice or iteration activity, the new initiating node is going to be the node representing the decision of the inserted activity. A new Type 2 realizability defect occurs on the newly-introduced initiating, participant-activated node n if some of its acting participants are neither immediately nor eventually aware of some of the other initiating nodes n_1, \dots, n_k ; formally:

$$\exists i \in [1, k] \exists p \in \text{actingParticipants}(n) : \text{AW}_{\text{fired}}(n_i, p) \notin \{\text{ia}, \text{ea}\}$$

Invariant 9. The application of an insertion change operator that adds a new initiating node n does not result in a new Type 2 realizability defect on n if and only if, given the other initiating nodes n_1, \dots, n_k , the following holds:

$$\forall i \in [1, k] \forall p \in \text{actingParticipants}(n) : \text{AW}_{\text{fired}}(n_i, p) \in \{\text{ia}, \text{ea}\}$$

Similarly, depending on the participant awareness resulting from traversing the new initiating node n , the acting participants of pre-existing initiating nodes may not be immediately or eventually aware of it, which would lead to Type 2 realizability defects affecting some or all the pre-existing initiating nodes; formally:

$$\exists i \in [1, k] \exists p \in \text{actingParticipants}(n_i) : \text{AW}_{\text{fired}}(n, p) \notin \{\text{ia}, \text{ea}\}$$

Invariant 10. The application of an insertion change operator that adds a new initiating node n does not result in a new Type 2 realizability defect on n if and only if, given the other initiating nodes n_1, \dots, n_k , the following holds:

$$\forall i \in [1, k] \forall p \in \text{actingParticipants}(n) : \text{AW}_{\text{fired}}(n, p) \in \{\text{ia}, \text{ea}\}$$

5.3.2.2 Type 2 Realizability Defects and Update Change Operators

Update change operators do not modify which participant-activated nodes are initiating, but can modify their acting participants and thus the participant awareness resulting from the traversing of those nodes. As in the case of insertion change operators, new Type 2 realizability defects can be introduced both on the initiating node being modified as well as on other nodes. The update change operators involved in these issues are the same considered in Section 5.3.1.2 for the introduction of Type 1 realizability defects (which is not surprising given the similarities between Type 1 and Type 2 realizability defects), namely:

- Message exchange activity:
 - Change Sender of Message Exchange Activity
 - Add Recipient to Message Exchange Activity
 - Remove Recipient from Message Exchange Activity
- Opaque activity:
 - Add Participant to Opaque Activity
 - Remove Participant from Opaque Activity
- Choice activity:
 - Change Decision Maker of Choice Activity
- Iteration activity:
 - Change Decision Maker of Iteration Activity

Given the initiating node n out of n_1, \dots, n_k , a new Type 2 realizability defect occurs on n if some of its acting participants are neither immediately nor eventually aware of some of the other initiating nodes n_1, \dots, n_k ; formally:

$$\exists n' \in \{n_1, \dots, n_k\} \setminus \{n\} \exists p \in \text{actingParticipants}(n) : \text{AW}_{\text{fired}}(n', p) \notin \{\text{ia}, \text{ea}\}$$

Invariant 11. The application of an update change operator that modifies the initiating node n does not result in a new Type 2 realizability defect on n if and only if, given the initiating nodes n_1, \dots, n_k , the following holds:

$$\forall n' \in \{n_1, \dots, n_k\} \setminus \{n\} \forall p \in \text{actingParticipants}(n) : \text{AW}_{\text{fired}}(n', p) \in \{\text{ia}, \text{ea}\}$$

From the formulation of Invariant 11, it is clear that only update change operators that add new or change existing acting participants of the modified node can result in a new Type 2 realizability defect. Specifically, this means that, out of the list of update change operators presented above, the “Add Recipient to Message Exchange Activity,” “Remove Recipient from Message Exchange Activity” and “Remove Participant from Opaque Activity” change operators *cannot violate* Invariant 11.

In addition to a new Type 2 realizability defect on the modified initiating node, update change operators applied to an initiating node can also cause new Type 2 realizability defects on other existing, unmodified initiating nodes. In fact, depending on the participant awareness resulting from traversing the updated initiating node n , the acting participants of pre-existing initiating nodes may not be immediately or eventually aware of it; formally:

$$\exists n' \in \{n_1, \dots, n_k\} \setminus \{n\} \exists p \in \text{actingParticipants}(n') : \text{AW}_{\text{fired}}(n, p) \notin \{\text{ia}, \text{ea}\}$$

Invariant 12. The application of an update change operator that modifies the initiating, participant-activated node n does not result in new Type 2 realizability defects on other initiating nodes if and only if, given the initiating, participant-activated node n_1, \dots, n_k , the following holds:

$$\forall n' \in \{n_1, \dots, n_k\} \setminus \{n\} \forall p \in \text{actingParticipants}(n') : \text{AW}_{\text{fired}}(n, p) \in \{\text{ia}, \text{ea}\}$$

In this case, new Type 2 realizability defects would be caused by the *reduced* participant awareness for the acting participants of the unmodified initiating nodes. Therefore, since they cannot lead to reduced participant awareness, the “Add Recipient to Message Exchange Activity” and “Add Participant to Opaque Activity” change operators cannot cause the introduction of such Type 2 realizability defects.

5.3.2.3 Type 2 Realizability Defects and Deletion Change Operators

The application of a delete change operator to a choreography may result in one or more initiating nodes being removed. When an initiating node is removed from a choreography, one of the following two cases takes place:

- No previously non-initiating node becomes initiating;
- One or more previously non-initiating nodes become initiating.

In the first case, no new Type 2 realizability defect can be introduced, as the initiating nodes that remain do not undergo changes in the way they affect participant awareness. Therefore, if they caused no Type 2 realizability defects before the change, they are not going to cause any afterwards.

However, if some previously non-initiating nodes n'_1, \dots, n'_l become initiating because of the application of a deletion change operator, this may indeed result in new Type 2 realizability defects. Unfortunately it is not possible to predict in general which nodes will become initiating because the impact of a deletion change to the structure an Awareness Model is too dependent on the structure of the choreography. (At design time of the choreography, however, it is trivial through

empirical verification.) Given the participant-activated nodes that represent the nodes that become initiating, new Type 2 realizability defect are introduced under the same conditions applicable to newly introduced initiating nodes, see Section 5.3.2.1. This intuition is captured by the following Invariant 13 and Invariant 14, which correspond to Invariant 9 and Invariant 10 modulo the adaptation of the phrasing to deletion change operators.

Invariant 13. The application of a deletion change operator that turns the participant-activated node n into an initiating node does not result on a new Type 2 realizability defect on n if and only if, given the other initiating nodes n_1, \dots, n_k , the following holds:

$$\forall i \in [1, k] \forall p \in \text{actingParticipants}(n) : \text{AW}_{\text{fired}}(n_i, p) \in \{\text{ia}, \text{ea}\}$$

Invariant 14. The application of a deletion change operator that turns the participant-activated node n into an initiating node does not result in a new Type 2 realizability defect on n if and only if, given the other initiating nodes n_1, \dots, n_k , the following holds:

$$\forall i \in [1, k] \forall p \in \text{actingParticipants}(n) : \text{AW}_{\text{fired}}(n, p) \in \{\text{ia}, \text{ea}\}$$

5.3.3 How to Introduce new Type 3 Realizability Defects

Type 3 realizability defects occur when some acting participants cannot observe their actions transitioning from enactable to non-enactable due to throwing and propagation of exceptions. In Section 4.4.2, the following two steps have been introduced to detect Type 3 realizability defects:

1. Identify which participant-activated nodes n_1, \dots, n_k are interruptible by the node n_e representing an activity **throw** e , i.e., the actions they represent can transition from enactable to non-enactable because of the traversing of the node n_e ;
2. Verify that all the acting participants of the nodes n_1, \dots, n_k are immediately aware of the throwing of the exception e (i.e., of the traversing of the node n_e) so that none of them will be performed by their acting participants when non-enactable; formally:

$$\forall n \in \{n_1, \dots, n_k\} \forall p \in \text{actingParticipants}(n) : \text{AW}_{\text{fireable}}(n, p) = \text{ia}$$

On the choreography shown in Figure 5.9, which has no realizability defects, the throwing of an exception of type e may change from enactable to non-enactable the action by p_1 of dispatching the message of type m_1 to p_2 . However, this does not result in a Type 3 realizability defect because the sender of that message exchange is immediately aware of the exception of type e being thrown.

Section 5.3.3.1, Section 5.3.3.2 and Section 5.3.3.3 respectively treat how Type 3 realizability defects can be introduced by using insertion, update and deletion change operators.

5.3.3.1 Type 3 Realizability Defects and Insertion Change Operators

Given the exception-propagation node n_e , the application of an insertion change operator can result in the introduction of Type 3 realizability defects in the following two cases:

1. If it changes the awareness-dependee nodes of n_e so that one or more of the acting participants of one of the nodes n_1, \dots, n_k that are interruptible by n_e are not immediately aware of n_e being enacted;
2. If it adds new participant-activated nodes to the choreography so that they are interruptible by n_e and some of their acting participants are not immediately aware of n_e being enacted.

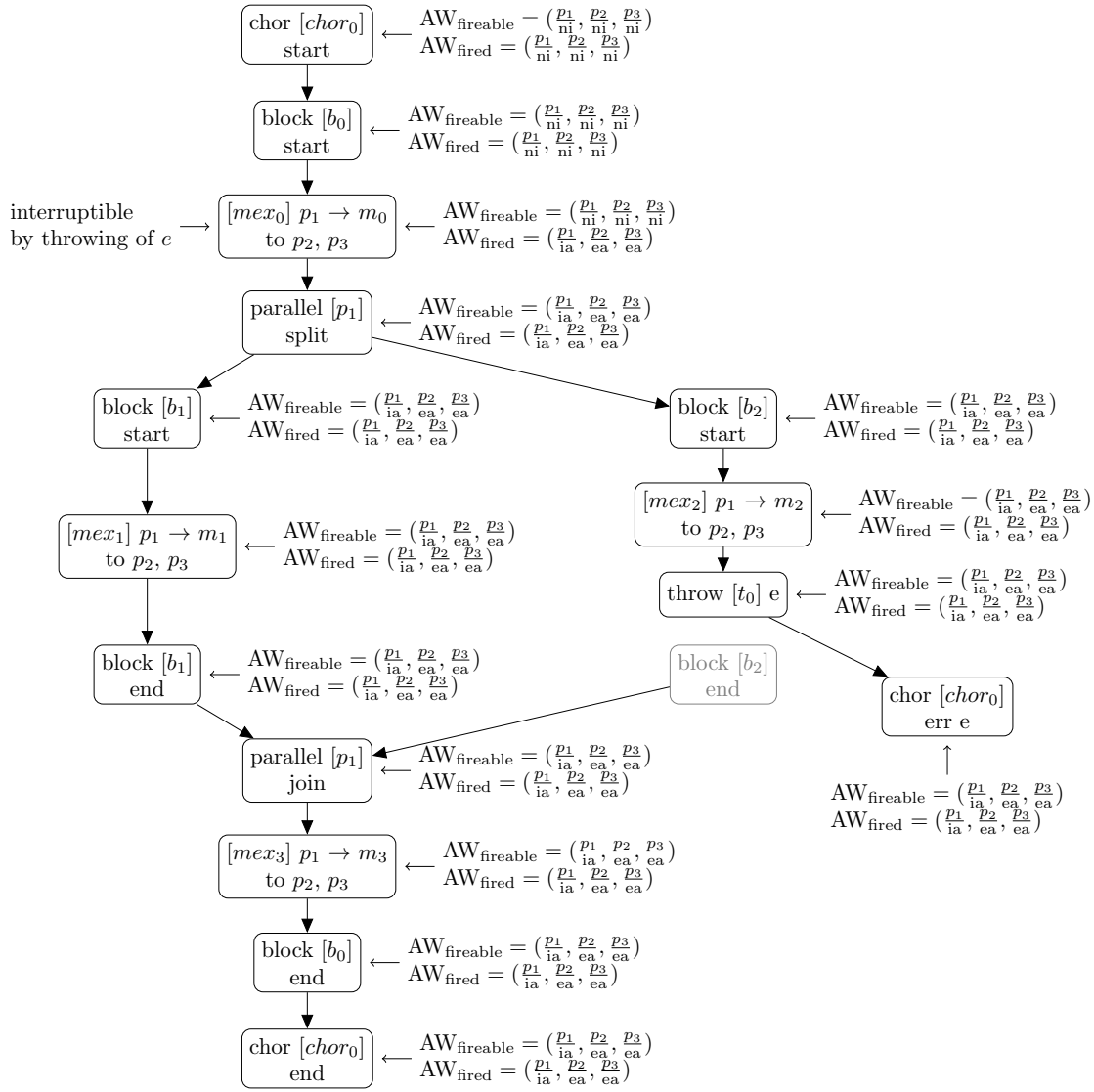
The first case is displayed in Figure 5.10, which shows the outcome of applying the “Insert Activity in a Block” change operator by adding the message exchange activity $[mex_4] p_3 \rightarrow m_4$ to p_1, p_2 before the **throw** $[t_0] e$ activity. After the change, the added participant-activated node mex_4 is the only awareness-dependee of the node t_0 representing the **throw** $[t_0] e$. The participant p_1 is

```

1 chor [chor0] ({
2   [mex0] p1 → m0 to p2, p3;
3   parallel [p1] do {
4     [mex1] p1 → m1 to p2, p3
5   } and {
6     [mex2] p1 → m2 to p2, p3;
7     throw [t0] e
8   };
9   [mex3] p1 → m3 to p2, p3;
10 })

```

(a) Source choreography.



(b) Awareness Model.

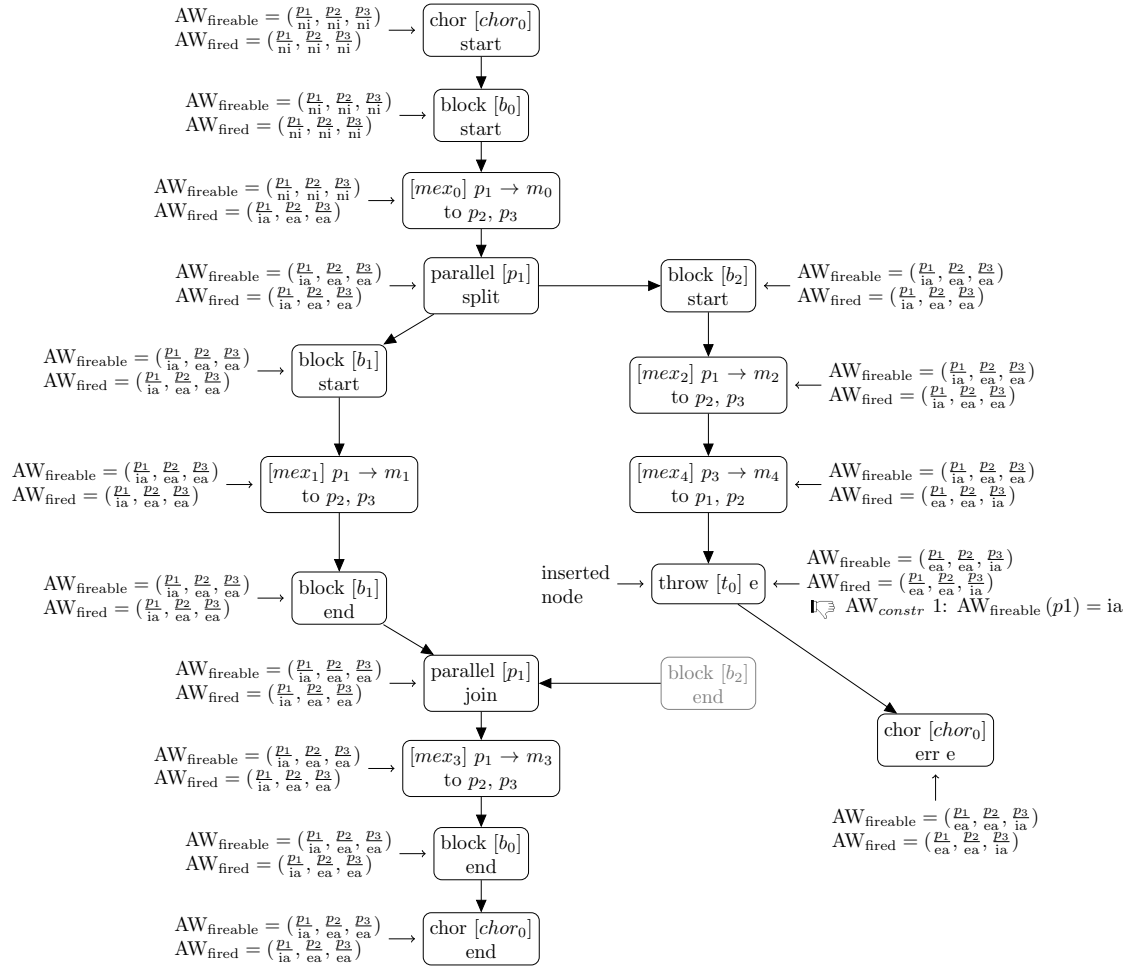
Figure 5.9: A strongly-realizable choreography with nodes that are interruptible by the throwing of the exception *e*.

```

1  chor [chor0] ({
2    [mex0] p1 → m0 to p2, p3;
3    parallel [p1] do {
4      [mex1] p1 → m1 to p2, p3
5    } and {
6      [mex2] p1 → m2 to p2, p3;
7    +   [mex4] p3 → m4 to p1, p2;
8      throw [t0] e
9    };
10   [mex3] p1 → m3 to p2, p3;
11 })

```

(a) Source choreography.



(b) Awareness Model.

Figure 5.10: The choreography shown in Figure 5.9 modified by inserting the message exchange activity *mex*₄ before the throw activity *t*₀; since *p*₁ is no longer immediately aware of the throwing of the exception of type *e*, this choreography is not strongly realizable.

not immediately aware of the traversal of the mex_4 node: it is just a recipient of that message exchange, and therefore it is only eventually aware of it taking place. Since p_1 is the acting participant of the participant-activated node mex_1 on the other branch of the parallel activity and mex_1 is interruptible by t_0 , this results in the new Type 3 realizability defect on the t_0 node. Prevention of such cases is captured by the following invariant:

Invariant 15. The insertion of a participant-activated node n that is awareness-dependee to an exception-propagation node n_e does not introduce a new Type 3 realizability defect as long as all the acting participants of the participant-activated nodes n_1, \dots, n_k interruptible by n_e are immediately aware of the n_e node being traversed; formally:

$$\forall p \in \left(\bigcup_{i \in [1, k]} \text{actingParticipants}(n_i) \right) : \text{AW}_{\text{fireable}}(n_e, p) = \text{ia}$$

The second case in which the “Insert Activity in a Block” can introduce new Type 3 realizability defects is when it results in the addition of new participant-activated nodes in other branches of the parallel activity and the acting participants of those new participant-activated nodes are not immediately aware of the exception-propagation node. This scenario is exemplified in Figure 5.11: the insertion of the **opaque** $[o_1]$ (p_1, p_2, p_3) activity in the other branch of the parallel activity means that now also p_2 and p_3 are acting participants of a node that is interruptible by the traversal of t_0 . This results in two new Type 3 realizability defects on the node t_0 , one per participant. The following invariant prevents this scenario:

Invariant 16. The insertion of a participant-activated node n does not introduce a new Type 3 realizability defect as long as all the acting participants of the participant-activated are immediately aware of the exception-propagation nodes n_{e_1}, \dots, n_{e_k} nodes that may interrupt n .

5.3.3.2 Type 3 Realizability Defects and Update Change Operators

Given the exception-propagation node n_e , an update change operator can introduce new Type 3 realizability defects in the following two cases:

1. By changing an awareness-dependee node of n_e so that one or more of the acting participants of one of the nodes n_1, \dots, n_k that are interruptible by n_e , are not immediately aware of n_e being enacted;
2. By modifying the acting participants of a node n that is interruptible by n_e so that some of the resulting acting participants of n are not immediately aware of n_e being traversed.

The first scenario is exemplified in Figure 5.12, in which the message exchange activity mex_2 has been modified by swapping the previous sender p_1 with the recipient p_2 . (This particular modification to the choreography requires the application of three update operators, namely first removing p_2 as recipient, setting p_2 as sender, and adding p_1 back as recipient; but similar effects can be obtained with single update change operators on other choreographies, so the point of this section is still valid.) Due to this modification to the choreography, p_1 is no longer immediately aware of the traversal of the node t_0 . Since p_1 is still the acting participant of the participant-activated node that represents the message exchange activity $[mex_1] p_1 \rightarrow m_1$ to p_2, p_3 , and that node is interruptible by the traversal of t_0 , this causes the introduction of the Type 3 realizability defect on t_0 .

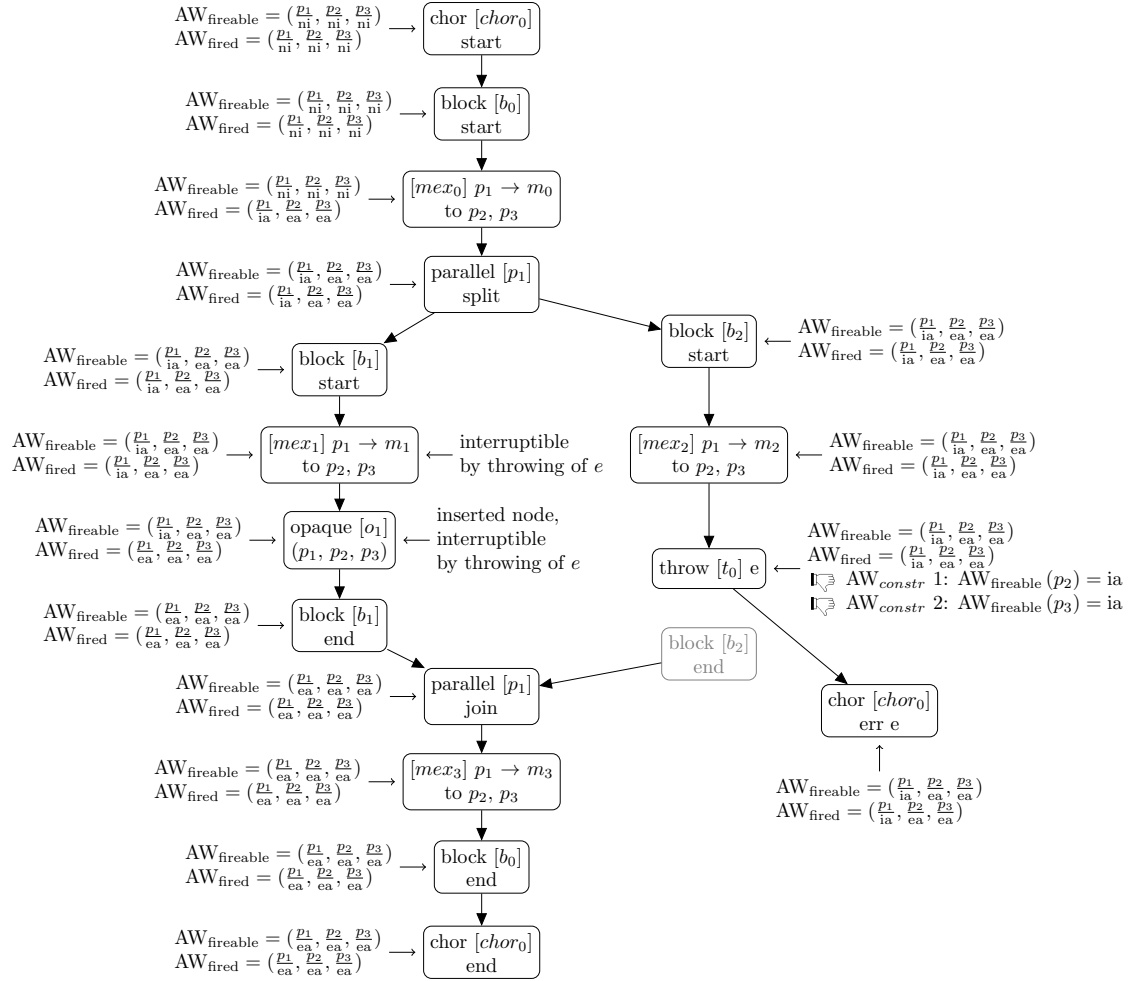
Invariant 17. An update change operator affecting an awareness-dependee of an exception-propagation node n_e does not introduce new Type 3 realizability defect as long as all the acting participants of the nodes that are interruptible by the n_{e_1}, \dots, n_{e_k} nodes are immediately aware of the traversal of n_{e_1}, \dots, n_{e_k} .

```

1  chor [chor0] ({
2    [mex0] p1 → m0 to p2, p3;
3    parallel [p1] do {
4      [mex1] p1 → m1 to p2, p3;
5  +   opaque [o1] (p1, p2, p3)
6    } and {
7      [mex2] p1 → m2 to p2, p3;
8      throw [t0] e
9    };
10   [mex3] p1 → m3 to p2, p3;
11 })

```

(a) Source choreography.



(b) Awareness Model.

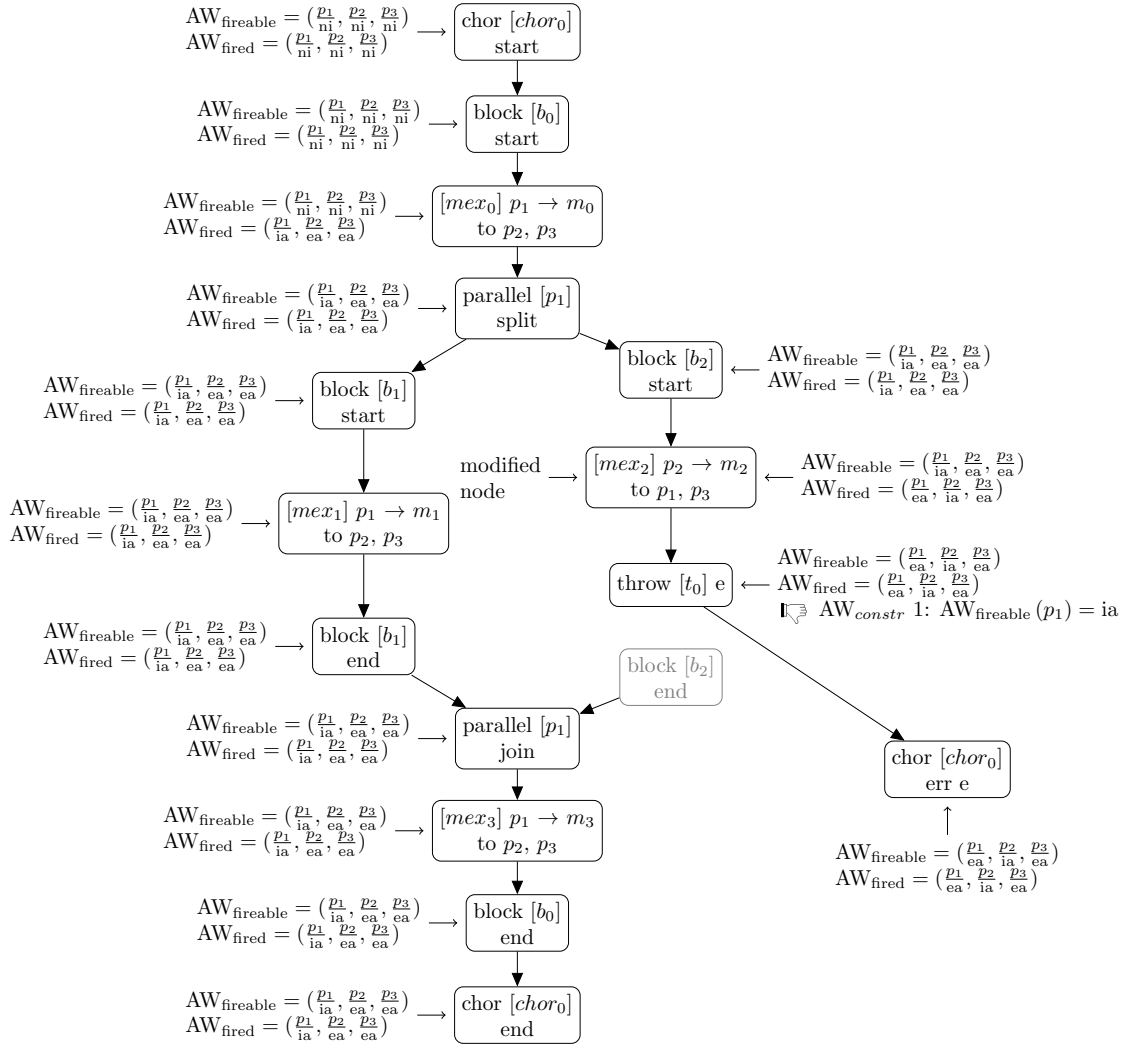
Figure 5.11: The choreography shown in Figure 5.9 modified by inserting the opaque activity o_1 in the left branch.

```

1 chor [chor0] ( {
2   [mex0] p1 -> m0 to p2, p3;
3   parallel [p1] do {
4     [mex1] p1 -> m1 to p2, p3
5   } and {
6     [mex2] p2 -> m2 to p1, p3;
7     throw [t0] e
8   };
9   [mex3] p1 -> m3 to p2, p3;
10 } )

```

(a) Source choreography.



(b) Awareness Model.

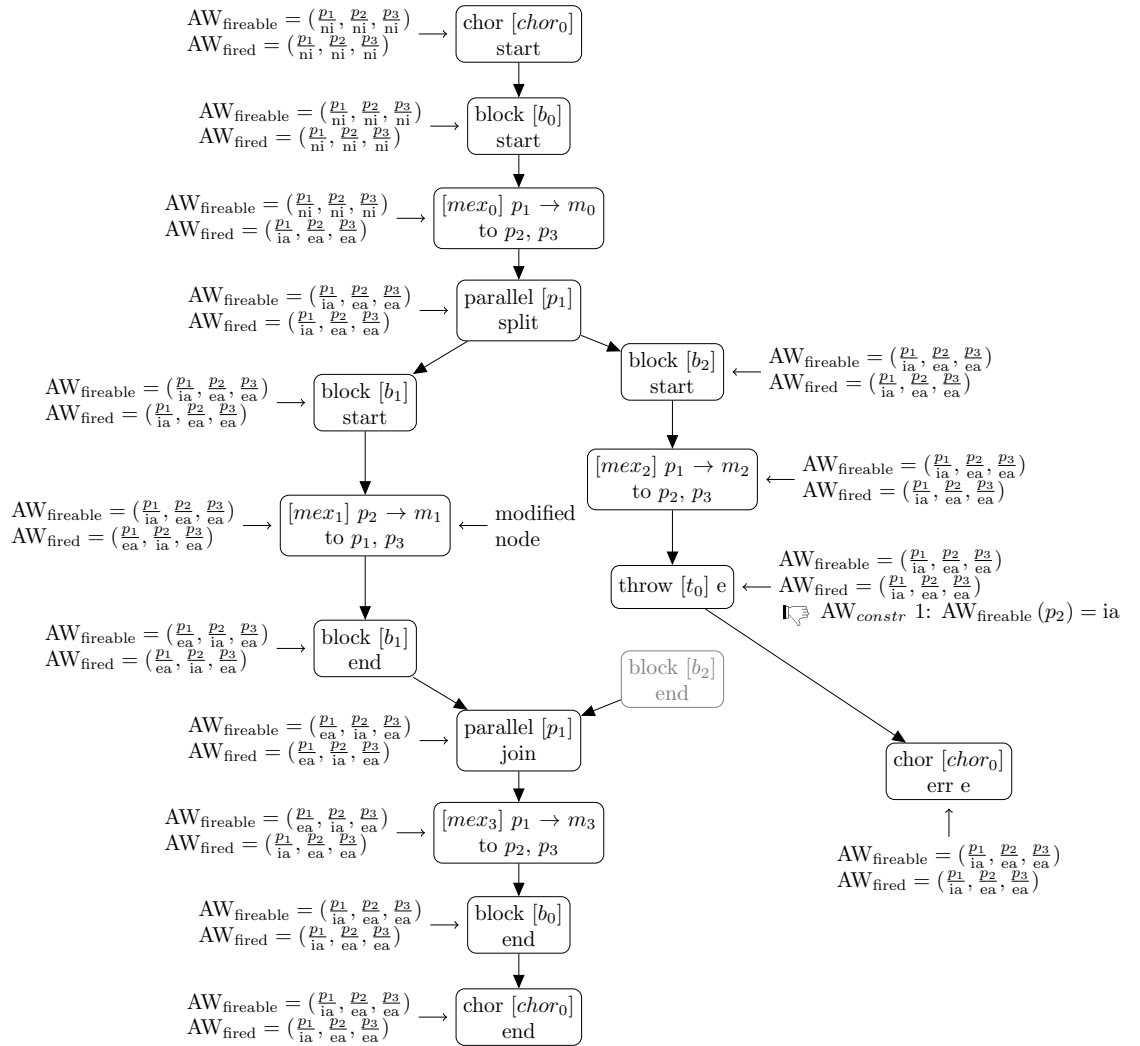
Figure 5.12: The choreography shown in Figure 5.9 modified by swapping the sender and a receiver in the message exchange activity *mex*₂.


```

1  chor [chor0] ({
2    [mex0] p1 → m0 to p2, p3;
3    parallel [p1] do {
4      ~ [mex1] p2 → m1 to p1, p3
5    } and {
6      [mex2] p1 → m2 to p2, p3;
7      throw [t0] e
8    };
9    [mex3] p1 → m3 to p2, p3;
10  })

```

(a) Source choreography.



(b) Awareness Model.

Figure 5.13: The choreography shown in Figure 5.9 modified by swapping the sender and a receiver in the message exchange activity mex_1 .

The other way in which update change operators can introduce new Type 3 realizability defects is by changing the acting participants of the participant-activated nodes that are interruptible by n_e and some of the resulting acting participants is not immediately aware of the traversal of n_e . This case is exemplified in Figure 5.13: this time, it is the message exchange activity mex_1 that has been modified by swapping the previous sender p_1 with the recipient p_2 .

Invariant 18. An update change operator that affects the participant-activated node n that can change from enactable to non-enactable due to the traversal of an exception-propagation node n_e does not introduce new Type 3 realizability defects as long as the resulting acting participants of n are all immediately aware of n_e .

5.3.3.3 Type 3 Realizability Defects and Deletion Change Operators

The application of deletion change operators can introduce new Type 3 realizability defects when they result in a change of the participant-awareness of the exception-propagation node n_e so that some of the acting participants of nodes interruptible by n_e are no longer immediately aware. This is the case when the delete change operator removes one activity that specifies in the Awareness Model some of the awareness-dependees of the throw node n_e . Therefore, the “shift” in awareness dependencies due to the deletion (see Section 5.3.1.3) may result in an new unsatisfactory awareness dependee for n_e .

Invariant 19. A delete change operator removing one or more awareness-dependees of the exception-propagation node n_e does not result in new Type 3 realizability defects if the resulting awareness-dependees are such that all acting participants of nodes interruptible by n_e are immediately aware of the traversal of n_e .

5.4 Remediation Strategies

This section presents remediation strategies to solve the three types of realizability defects that can affect ChorTex choreographies. The types of realizability defects are tackled separately: Type 1 realizability defects are discussed in Section 5.4.1, Type 2 in Section 5.4.2 and Type 3 in Section 5.4.3. Each of these sections starts by outlining one or more *abstract remediation strategies* for solving the investigated type of realizability defect. Abstract remediation strategies are “sketches” of approaches that can be used to fix the realizability defects, which are then refined into the “actual” remediation strategies that are used to generate remediation plans.

Additionally to the remediation strategies focused on the various types of realizability defects, there is also one remediation strategy that applies for all types of realizability defects: modify the choreography so that the node affected by the realizability defect is removed. This remediation strategy, which is tantamount to “removing the tooth that aches,” is discussed in Section 5.4.4.

We believe the framework of remediation strategies proposed in the remainder of this section to be *complete*, i.e., at least one remediation plan can always be generated for a given realizability defect. It should be noted, though, that we are unfortunately not aware of any way to formally proving our claim. However, our belief is supported by the fact that no counter-example has surfaced in the extensive experimentation and modeling practice performed with the prototype.

5.4.1 Remediating Type 1 Realizability Defects

In Section 5.3.1 is discussed how Type 1 realizability defects are the result of insufficient participant awareness by some acting participants with respect to the enactability of their actions. From a very high-level perspective, Type 1 realizability defects can be solved with the following approach:

Abstract Remediation Strategy 1. A realizability defect of Type 1 on the participant-activated node n_a can be corrected by modifying the choreography so that all acting participants of n_a are immediately or eventually aware of the traversing of all awareness dependee nodes of n_a .

As its moniker suggests, the Abstract Remediation Strategy 1 is too generic to be translated directly into remediation plans. In the remainder of this section we consider two following “specializations” of the Abstract Remediation Strategy 1.

Remediation Strategy 1 (Adjust the acting participants of the affected node). Given the Type 1 realizability defect affecting the participant p on the node n_a , remove p as acting participant of n_a and, if necessary, replace it with other participants that are aware of n_a becoming traversable.

This remediation strategy can be explained as follows: make the node affected by the Type 1 realizability defect “fit” its awareness-dependee nodes, which remain unchanged.

Remediation Strategy 2 (Adjust unsatisfactory awareness dependees of the affected node). Given the Type 1 realizability defect affecting the participant p on the node n_a , modify the choreography so that p is aware of the traversing of all awareness dependees of n_a .

The Remediation Strategy 2 is fundamentally the opposite of the Remediation Strategy 1: change the awareness dependees of n_a so that they “fit” the participant awareness requirements.

The application of Remediation Strategy 1 and Remediation Strategy 2 to generate remediation plans is treated in Section 5.4.1.1 and Section 5.4.1.2, respectively.

5.4.1.1 Adjust the Acting Participants of the Affected Node

The Remediation Strategy 1 fundamentally consists in “making do” with the awareness dependees of the node affected by the Type 1 realizability defect. In a nutshell, this remediation strategy consists of the following:

1. Check which participants *could* be the acting participants of the participant-activated node n_a without that resulting in a realizability defect of Type 1; assume that those participants are p_1, \dots, p_m ;
2. Modify, by means of update change operators, the source activity of n_a as to have a subset of p_1, \dots, p_m as acting participants.

The first step is straightforward: the participants that could be acting participants of the node n_a without this resulting in one or more Type 1 realizability defects are those that are immediately or eventually aware of n_a becoming enactable, i.e., those participants p_1, \dots, p_m so that:

$$\{p_1, \dots, p_m\} := \left\{ p : \text{AW}_{\text{fireable}}(n_a, p) \in \{\text{ia}, \text{ea}\} \right\}$$

If there are no participants that are either immediately or eventually aware of the node n_a , there is no way of modifying just n_a and solve the Type 1 realizability defect affecting it, and this strategy is not applicable. In the remainder we assume that there are some “eligible” participants p_1, \dots, p_m that are *at least* eventually aware of n_a becoming traversable. (Remember that immediate-awareness implies eventual-awareness, see Section 4.5.)

By means of its constructs, ChorTex allows to specify four types of participant-activated nodes, namely those representing dispatch of messages, the enactment of opaque activities and decisions in the scope of choice and iteration activities (see Definition 3.1). These participant-activated nodes result from only four types of activities, namely message exchange activities, opaque activities, choice activities and iteration activities. Since our goal is to modify n_a to “fit” its awareness dependee nodes, the change operators that we can use to this end are only the update ones. Depending on the type of the source activity of the node n_a , the update change operators that can be applied are the following:

- Message exchange activity:
 - Change Sender of Message Exchange Activity”

- Opaque activity:
 - Remove Participant from Opaque Activity”
- Choice activity:
 - Change Decision Maker of Choice Activity”
- Iteration activity:
 - Change Decision Maker of Iteration Activity”

In the remainder we discuss separately each of the four types of activities listed above with respect to how to change the acting participants of n_a to be a subset of the participants p_1, \dots, p_m .

Message exchange activity: Assume the following message exchange activity:

$$[mex] p_s \rightarrow m \text{ to } p_{r_1}, \dots, p_{r_t}$$

In the Awareness Model, this message exchange activity is mapped to a participant-activated node n_a that has the sender p_s as its only acting participant (see Definition 3.1). The Type 1 awareness constraint associated with the participant-activated node n_a representing **a** in the AWM is the following:

$$AW_{\text{fireable}}(n_a, p_s) \in \{\text{ia}, \text{ea}\}$$

The assumption that there is a Type 1 realizability defect for p on the node n_a means that the above awareness constraint is unsatisfied. That is, the participant awareness of p with respect to **a** becoming enactable has as value either unaware or not-involved. Changing the sender of the message exchange activity is the only opportunity we have to “align” the acting participants of **a** with a subset of the participants p_1, \dots, p_m that are aware of **a** becoming enactable. Specifically, we must use the “Change Sender of Message Exchange Activity” change operator to replace p_s with one of p_1, \dots, p_m . If there are multiple eligible participants, one remediation plan is generated for each participant.

In order to respect the “On Introducing new Realizability Defects” principle, the eligible participants needs to be further vetted as not to introduce new realizability defects. Since this remediation strategy is based on the “Change Sender of Message Exchange Activity” change operator, the resulting remediation plans must additionally satisfy Invariant 5, Invariant 6, Invariant 11, Invariant 12, Invariant 17 and Invariant 18 (see Table 5.1).

Even when satisfying all the constraints listed above, some participants may of course be more “appropriate” than others to become the new sender, for example because of the information they have access to, which might be necessary to compose the message, or the roles they play in the inter-organizational business process underpinning the choreography. However, these considerations are outside the scope of the present thesis and are left as future work (see Section 7.3).

It should be noted that changing the sender of a message exchange activity with another participant may compromise the well-formedness of the choreography by violating the “Sender is not a Recipient” well-formedness requirement. In this case, an additional update change is required to be performed before the sender can be changed: the removal of the new sender p' as recipient. This is performed via the “Remove Recipient from Message Exchange Activity” change operator, which requires the resulting remediation plan to additionally satisfy Invariant 6 and Invariant 12.

Moreover, if p' is currently the *only* recipient of the message exchange, this would violate the syntax of ChorTex, which requires at least one recipient for each message exchange activity (see Section 3.2). Therefore, if p' is made sender and removed as the only recipient, another participant must take its place as recipient. If adding a new recipient is required, multiple, alternative remediation plans are generated, each adding a different, already-declared participant (adding completely new participants does not seem to make much sense from the modeling perspective, but it is of course also a possibility). As shown in Table 5.1, the “Add Recipient to Message Exchange Activity” update change operator cannot introduce any new Type 1, Type 2 or Type 3 realizability defect.

Opaque activity: Assume the following opaque activity:

$$\text{opaque } [o] \ (p_1, \dots, p_m)$$

This opaque activity is mapped in the Awareness Model to a participant-activated node n_a with p_1, \dots, p_m as acting participants (see Definition 3.1). A realizability defect of Type 1 on the node representing this activity means that one of the acting participants is neither immediately nor eventually aware of the node n_a becoming traversable. (If more than just one of the acting participants p_1, \dots, p_m are not aware of n_a becoming traversable, there are correspondingly many Type 1 realizability defects affecting the node n_a .) Assume p out of p_1, \dots, p_m to be the acting participant for which occurs the Type 1 realizability defect that we aim at correcting. To solve this Type 1 realizability defect by updating the opaque activity, p must be removed from the opaque activity using the “Remove Participant from Opaque Activity” change operator.

As shown in Table 5.1, the removal of p as acting participant must satisfy Invariant 6 and Invariant 12 in order not to introduce new Type 1 and Type 2 realizability defects, respectively. Additionally, removing p from the opaque activity may violate the syntax of ChorTex (opaque activities must have at least two participants, see Section 3.2). In this case, more participants must be added to the opaque activity by means of the “Add Participant to Opaque Activity” change operator, which in turn must satisfy Invariant 5, Invariant 11 and Invariant 18.

It is of course possible that no participant declared in the choreography fits these requirements, in which case this remediation strategy is not applicable. In case multiple participants fit these requirements, alternative remediation plans are generated, one per participant to be added to the opaque activity.

Choice activity: Assume the following choice activity:

$$\text{choice } [c] \ p \ \text{either } \mathbf{A}_1 \ \text{or } \dots \ \text{or } \mathbf{A}_l$$

Notice that, while the branches of the choice activity may contain activities that are mapped to any number of participant-activates nodes, the only participant-activated node of generated *directly* from the choice activity is the participant-activated node n_a that represent the decision taken by the decision maker p (see Definition 3.1). If the node n_a is affected by a realizability defect of Type 1, then it is necessarily the case that p is neither immediately nor eventually aware of n_a becoming traversable. The current decision maker p must be replaced with another participant p' using the “Change Decision Maker of Choice Activity” change operator. The participants eligible to become the new decision maker are those that are either eventually or immediately aware of n_a being traversable (otherwise, we would be simply replace one Type 1 realizability defect with another). The update change operator that replaces p with p' must satisfy Invariant 5, Invariant 6, Invariant 11, Invariant 12, Invariant 17 and Invariant 18 (see Table 5.1). It is of course possible that no participant is immediately nor eventually aware of n_a becoming enactable and that also satisfies the invariants listed above, in which case this remediation strategy is not applicable. In case multiple participants fit these requirements, alternative remediation plans are generated, one per eligible participant, to set the respective participant as decision maker of the choice activity.

Iteration activity: The case of iteration activity is practically identical to the previous case of choice activity all the way to the invariants have to be verified by the generated remediation plans. The only difference that the change operator used is the “Change Decision Maker of Iteration Activity” instead of “Change Decision Maker of Choice Activity.”

5.4.1.2 Adjust Unsatisfactory Awareness Dependees of the Affected Node

The Remediation Strategy 2 consists in modifying the choreography so that the unsatisfactory awareness dependees n_1, \dots, n_k of the node n_a become satisfactory. All unsatisfactory awareness dependees must be “treated” within the scope of one single remediation plan, otherwise the Type 1

realizability defect would not be fixed. We propose the following three options to deal with each unsatisfied awareness dependee:

- Inject adapting awareness dependees;
- Fix unsatisfactory awareness dependees;
- Delete unsatisfactory awareness dependees.

The changes resulting by the strategies above must be composed in a composite remediation plan; the order in which the change operators are sequenced is irrelevant, because none of these options can have side-effects on the other awareness dependees of n_a . The three options listed above to deal with unsatisfactory awareness dependees are presented separately in the remainder of this section.

Inject adapting awareness dependees: The goal is to replace the unsatisfactory awareness dependee n' of the node n_a that represents the action a by “inserting” a new participant-activated node n'' that lays on every path connecting n' and n_a . That is, n'' replaces n' as awareness dependee of n_a and becomes itself an awareness dependent node of n' . In other words, n'' behaves as an “adapter” for the participant awareness that flows from n' to n_a . The insertion of n'' is accomplished by using the “Insert Activity in a Block” change operator to add a new message exchange activity to the choreography so that:

1. The node n'' that represents the newly inserted message exchange activity in the Awareness Model lies on all paths connecting n' and n_a that do not contain other participant-activated nodes (otherwise, n'' would not replace n' as awareness dependee of n_a);
2. All the acting participants of n_a are either immediately or eventually aware of the performing of n'' so that n'' , as awareness dependee, is satisfactory for n_a . That is, the traversal of n'' must affect the participant awareness so that all the acting participants of n_a are immediately or eventually aware of the performing of the action represented by n'' ; put formally:

$$actingParticipants(n_a) \subseteq \{p : AW_{\text{fired}}(n'', p) \in \{ia, ea\}\}$$

This provision guarantees that the Type 1 realizability defect realizability defect currently diagnosed on n_a is remediated by the introduction of the new activity.

As already noted in Section 5.3.1, the syntax of ChorTex restrains where new message exchange activities can be inserted to simply blocks. This provides us with a straightforward option for where to insert the new message exchange activity: right “before” the source activity of n_a in the parent block of the latter, which guarantees us that n' dominates n_a and that, therefore, n'' replaces n' as awareness dependee of n_a .

Since this remediation strategy uses the “Insert Activity in a Block” change operator, the result-change must satisfy Invariant 1, Invariant 2, Invariant 3, Invariant 4, Invariant 9, Invariant 10, Invariant 15 and Invariant 16 (see Table 5.1).

Fix Unsatisfactory Awareness Dependee: In this scenario, given the unsatisfactory awareness dependee n' , the goal is to modify the source activity of n' by means of update change operators so that n' is no longer an unsatisfactory awareness dependee of n_a . This approach is almost identical to the one presented in Section 5.4.1.1, except that the awareness dependee node n' is modified instead of the node n_a that is affected by the Type 1 realizability defect. Since n' is a participant-activate node (all awareness dependee are by definition, see Definition 5.2), its source activity must be either a message exchange activity, an opaque activity, a choice activity or an iteration activity; we discuss the four types separately:

Message Exchange Activity: The source activity of n' is a message exchange activity. Since n' is an unsatisfactory awareness dependee of the node n_a affected by the Type 1 realizability

defect, the acting participant p of n_a is neither immediately nor eventually aware of the traversing of n' . The solution is straightforward: add p as recipient of the message exchange activity by means of the “Add Recipient to Message Exchange Activity” update change operator. Notice that, by doing so, we can violate neither the “Sender is not a Recipient” nor the “Distinct Recipients in Message Exchange Activities” well-formedness requirements because the participants that are added as recipients are not already involved in the message exchange activity. Moreover, since the “Add Recipient to Message Exchange Activity” cannot violate any invariant (see Table 5.1), this change cannot introduce Type 1, Type 2 or Type 3 realizability defects.

Opaque Activity: In this case, the source activity of n' is an opaque activity. Similarly to the previous case, the solution is to add the participant p to the opaque activity by means of the “Add Participant to Opaque Activity” update change operator. Also similarly to the case of the message exchange activity, the “Distinct Participants in Opaque Activities” well-formedness requirement cannot be violated by this change because the participant p that is added was not already involved in the opaque activity (otherwise it would be already eventually aware of its completion, see Section 4.3.3). Since the “Add Participant to Opaque Activity” update change operator is being applied, the resulting change must satisfy Invariant 5, Invariant 11 and Invariant 18 (see Table 5.1).

Choice Activity: If the source activity of the unsatisfactory awareness dependee n' of n_a is a choice activity, the only update change operator that we can use to turn n' into a satisfactory awareness dependee of n_a is the “Change Decision Maker of Choice Activity” change operator. Interestingly, the applicability of this approach is restricted by the type of the source activity of the node n_a . Since choice activities have exactly one acting participant (the decision maker), this strategy can be applied if and only if there is exactly one acting participant for n_a ; otherwise, we would trade a Type 1 realizability defect with another. Only three out of the four types of actions have always exactly one acting participant, namely the dispatching of messages and decisions in the scope of both choice and iteration activities. Opaque activities, instead, have always at least two acting participants. Therefore the current strategy cannot be applied if the source activity of the node n_a is an opaque activity. Assuming that source of n_a is not an opaque activity, the awareness dependee n' of n_a is made satisfactory by changing the decision maker of the choice activity a' with the only acting participant of n_a . Since the “Change Decision Maker of Choice Activity” change operator is being applied, the resulting change must satisfy Invariant 5, Invariant 6, Invariant 11, Invariant 12, Invariant 17 and Invariant 18 (see Table 5.1).

Iteration Activity: This case is almost identical to the one of choice activities, with the only difference being that the update change operator to be applied is the “Change Decision Maker of Iteration Activity” one.

Delete Unsatisfactory Awareness Dependee: This last approach to fixing the unsatisfactory awareness dependee n' of n_a consists in deleting the source activity of n' . Provided that the source activity of n_a is not nested into the source activity of n' , this case is equivalent to the deletion of the source activity of n_a discussed in Section 5.4.4. If, instead, the source activity of n_a is indeed nested into the source activity of n' , then the removal of the first would also cause the removal of the latter, which is not the goal of this remediation strategy; in such a case, this remediation strategy is not applicable.

5.4.2 Remediating Type 2 Realizability Defects

If a choreography has only one initiating node, no awareness constraints are associated with the one initiating node and therefore in that choreography there cannot be Type 2 realizability defects

(see Section 4.4.1.2). Thus, in the present section we assume that the choreography has multiple initiating nodes.

In a choreography with multiple initiating nodes, each initiating node n is associated with an awareness constraint that requires that every acting participant of the other initiating nodes is either immediately or eventually aware of the traversing of n . If the choreography has a Type 2 realizability defect associated with the initiating node n , it means there is participant p that is not aware of the traversal of n and that p is also an acting participant of one or more of the initiating nodes other than n . This observation leads to the following abstract remediation strategy:

Abstract Remediation Strategy 2. The choreography must be changed so that all the acting participants of every initiating node are either immediately or eventually aware of the traversing of all the other initiating nodes.

We foresee the following remediation strategies for creating remediation plans that solve Type 2 realizability defects:

Remediation Strategy 3 (Adjust affected initiating node). Given the Type 2 realizability defect affecting the participant p on the initiating node n , modify n so that the participant p is aware of its traversal.

Remediation Strategy 4 (Introduce a new initiating node). Given the Type 2 realizability defect affecting the participant p on the initiating node n , modify the choreography so that the node n is no longer initiating.

Remediation Strategy 5 (Fix the other initiating nodes). Given the Type 2 realizability defect affecting the participant p on the initiating node n , modify the other initiating nodes so that p is no longer an acting participants in any of them.

The Remediation Strategies 3, 4 and 5 are treated in Section 5.4.2.1, 5.4.2.2 and 5.4.2.3, respectively.

5.4.2.1 Adjust Affected Initiating Node

The Remediation Strategy 3 consists of “making do” with whichever participants are aware of the traversing of the other initiating nodes. That is, this strategy consists in modifying the source activity a of the affected initiating node n so that its resulting acting participants are a subset of those that are aware of the traversing of all other initiating nodes. How this is accomplished by means of update change operators; the specific update change operators to be applied depend on the type of the source activity a , which can be either a message exchange, opaque, choice or iteration activity. These different activity types are discussed separately in the remainder, in which we also assume p_1, \dots, p_n to be the participants that are aware of the traversal of all the initiating nodes n_1, \dots, n_m other than n , namely:

$$\{p_1, \dots, p_n\} := \{p : \forall i \in [1, m] : \text{AW}_{\text{fired}}(p, n_i) \in \{\text{ia}, \text{ea}\}\}$$

Notice that, if there is no participant that is aware of the traversal of all the other initiating nodes, this remediation strategy is not applicable.

Message exchange activity: The sender of the message exchange activity is not aware of the traversal of one or more of the other initiating nodes. The solution, in this case, is replacing the sender p with another participant chosen among p_1, \dots, p_n using the “Change Sender of Message Exchange Activity” update change operator. Since we are applying the “Change Sender of Message Exchange Activity” update change operator, the resulting change must satisfy Invariant 5, Invariant 6, Invariant 11, Invariant 12, Invariant 17 and Invariant 18 (see Table 5.1). Of course, changing the sender of the message exchange activity with another participant may compromise the well-formedness of the choreography by violating the “Sender is not a Recipient” well-formedness requirement. In this case, the additional changes to be applied to the message exchange activity are in every aspect equivalent to those described in Section 5.4.1.1.

Opaque activity: The participant p of the opaque activity that is source of the affected initiating node n must be removed from the activity using the “Remove Participant from Opaque Activity” change operator. As shown in Table 5.1, the removal of p as acting participant must satisfy Invariant 6 and Invariant 12 in order not to introduce new realizability defects. Additionally, removing p from the opaque activity may violate the syntax of ChorTex (opaque activities must have at least two participants, see Section 3.2). In this case, more participants must be added to the opaque activity by means of the “Add Participant to Opaque Activity” as described in Section 5.4.1.1.

Choice activity: The current acting participant of the affected initiated node is replaced with p using the “Change Decision Maker of Choice Activity.” The resulting change must satisfy Invariant 5, Invariant 6, Invariant 11, Invariant 12, Invariant 17 and Invariant 18 (see Table 5.1). This type of change cannot compromise the well-formedness of the choice activity, therefore additional changes are not required.

Iteration activity: The case of iteration activity is identical to the previous case of choice activity, except that the “Change Decision Maker of Iteration Activity” change operator is applied.

5.4.2.2 Introduce a New Initiating Node

The idea underpinning the Remediation Strategy 4 is to remediate the Type 2 realizability defect affecting the participant p on the initiating node n by changing the choreography so that n is no longer initiating (and therefore there cannot be any longer Type 2 realizability defects associated with it). This is achieved by modifying the choreography so that is introduced a new initiating node n' that dominates n ; since n' dominates n , n cannot be any longer initiating: all the paths connecting the start node with n pass through n' . The new node n' is the result of the insertion, by means of the “Insert Activity in a Block” change operator, of a message exchange activity a' before the source activity a of n . The sender and recipients of the message exchange activity a' are selected so that they collectively form the set of all acting participants of the other initiating nodes (otherwise n' would be affected by Type 2 realizability defects itself) and of all the acting participants of n (otherwise n would become affected by a Type 1 realizability defect). To the end of this remediation strategy, it is not important which participant will be the sender of the newly-introduced message exchange activity and which the recipients: for each combination, a different remediation plan is generated. Since the “Insert Activity in a Block” change operator is being applied, the remediation plans must verify Invariant 2, Invariant 3, Invariant 4, Invariant 9, Invariant 10, Invariant 15 and Invariant 16.

Potentially, opaque activities could also be used to implement this strategy. In this case, the acting participants of the newly-inserted opaque activity would be the set of acting participants of the other initiating nodes. However, given the underspecification inherent in opaque activities (see Section 3.2), introducing new message exchanges seems a much more meaningful way of modifying the choreography.

5.4.2.3 Fix the Other Initiating Nodes

Given a Type 2 realizability defect affecting the participant p on the node n , the Remediation Strategy 5 consists of modifying the choreography so that p is no longer an acting participant of any initiating node (and, therefore, the fact that p is not aware of n no longer causes a Type 2 realizability defect). For each other initiating node n' of the choreography, one of the following approaches is applied:

Fix the other initiating node: The choreography is modified so that p is no longer an acting participant of n' ;

Introduce a new initiating node: A new initiating node is introduced that replaces n' as initiating node and that does not have p as acting participant;

Delete the other initiating node: The choreography is modified so that the node n' is removed from the Awareness Model.

The first two of the approaches listed above follow the same logic of the remediation strategies presented in Section 5.4.2.1 and Section 5.4.2.2 and are therefore here omitted. Similarly, the third approach, namely the deletion of the initiating node n' , follows the same rules of the changes that cause the deletion of any node from the Awareness Model, which is covered in Section 5.4.4.

5.4.3 Remediating Type 3 Realizability Defects

Type 3 realizability defects are found in choreographies in which some of the acting participants cannot observe their actions being interrupted by the throwing of some exceptions (see Section 4.4.2). Generally, the correction of Type 3 realizability defects is achieved using the following abstract remediation strategy:

Abstract Remediation Strategy 3. The choreography must be changed so that all the acting participants of participant-activated nodes that can be interrupted by the traversal of the **throw** e node must be immediately aware of the latter being traversed.

In the remainder of this section we will refer to the node **throw** e as n_e . We foresee the following four possibilities to implement the Abstract Remediation Strategy 3:

Remediation Strategy 6 (Adjust participant-awareness of the throw node). Given the Type 3 realizability defect on the node n_e affecting the acting participant p of the interruptible node n , modify the choreography so that p is immediately aware of the traversal of n_e .

Remediation Strategy 7 (Adjust acting participants of the interruptible node). Given the Type 3 realizability defect on the node n_e affecting the acting participant p of the interruptible node n , modify the node n by removing p as acting participant and, if necessary, replace it with other participants that are immediately aware of the traversal of n_e .

Remediation Strategy 8 (Remove the interruptible node). Given the Type 3 realizability defect on the node n_e affecting the acting participant p of the interruptible node n , change the choreography as to remove the node n .

Remediation Strategy 9 (Remove the throw node). Given the Type 3 realizability defect on the node n_e affecting the acting participant p of the interruptible node n , change the choreography as to remove the node n_e .

Before discussing the above remediation strategies in detail, however, it is important to recall an observation offered in Section 4.7.2: *at most one participant is immediately aware of the traversal of any node*, including the nodes that represent the throwing of an exception. This observation limits the applicability of Remediation Strategy 6 and Remediation Strategy 7, which are discussed in Section 5.4.3.1 and Section 5.4.3.2, respectively. The Remediation Strategy 8 and Remediation Strategy 9, instead, are discussed in Section 5.4.3.3 and Section 5.4.3.4.

5.4.3.1 Adjust Participant-Awareness of the Throw Node

Given the Type 3 realizability defect on the node n_e affecting the acting participant p of n , the Remediation Strategy 6 foresees the modification of the choreography so that p becomes immediately aware of the traversal of the node n_e . Similarly to the cases of adjusting the participant-awareness of certain nodes already discussed for Type 1 and Type 2 realizability defects (see Section 5.4.1.2 and Section 5.4.2.3, respectively), this remediation strategy consists in modifying which nodes are awareness dependees of n_e and how their traversal affects the participant-awareness of p . In particular, for each awareness-dependee n' of n_e , one of the following options has to be applied:

Fix unsatisfactory awareness dependee: Modify the node n' so that p is immediately aware of its traversal;

Inject adapting awareness dependee: Modify the choreography so that new participant-activated nodes are “injected” between the n' and n_e ; these new nodes replace n' as awareness dependees of n_e and make sure that p is immediately aware of their traversal, which makes p also immediately aware of the traversal of n_e ;

Delete unsatisfactory awareness dependee: Remove the node n' .

In the remainder of this section, we discuss these three options separately.

Fix unsatisfactory awareness dependee: The aim of this approach is to modify the node n' , which is an awareness dependee of n_e , so that p is immediately aware of its traversal. How this is achieved depends on the type of the source activity a of n' .

If the source activity a of n' is a message exchange activity, then p must become its sender. This approach is equivalent to the one discussed in Section 5.4.1.2 for the “fix unsatisfactory awareness dependees” approach.

In the cases of a being either a choice or iteration activity, p must become the decision maker. Similarly to the case of a being a message exchange activity, dealing with this scenario is equivalent to the one discussed in Section 5.4.1.2 for the “fix unsatisfactory awareness dependees” approach.

If a is an opaque activity, however, this approach is not applicable. The reason is the following: by the definition of the function $f(n, p)$, which specify how the traversal of a node affects the participant-awareness (see Section 4.3.3), no participant is immediately aware of the traversal of a node representing an opaque activity. Therefore, it is not possible to update the source activity of the unsatisfactory awareness dependee node in order to make p immediately aware of its traversal.

Inject adapting awareness dependee: This approach is equivalent to its namesake discussed in Section 5.4.1.2 and it is therefore here omitted.

Delete unsatisfactory awareness dependee: Similarly to the previous case, this approach is equivalent to its namesake discussed in Section 5.4.1.2 and it is therefore here omitted.

5.4.3.2 Adjust Acting Participants of the Interrupted Node

Given the Type 3 realizability defect on the node n_e affecting the acting participant p of the node n , the Remediation Strategy 7 consists of changing the node n so that its acting participants are all immediately aware of the traversal of n_e . As already pointed out earlier, there can be at most one participant immediately aware of the traversal of n_e . If there are actually no participants immediately aware of the traversal of n_e , this remediation strategy is not applicable. In the remainder we assume that there is a participant p' immediately aware of the traversal of n_e .

The goal is to make p' the only acting participant of n , which is possible only if n represents the dispatching of a message (and p' has to become its sender) or the decision of a choice or iteration activity (and p' has to become the decision maker). These cases are treated in the same manner as discussed in Section 5.4.1.1.

In case the source activity of n is an opaque activity, however, this remediation strategy is not applicable: opaque activities have at least two participants, Since it is not possible for more than one participant to be immediately aware of the throw node n_e , there is no way to change the source activity of n so that all its acting participants are immediately aware of the traversal of n_e .

5.4.3.3 Remove the Interrupted Node

Given the Type 3 realizability defect on the node n_e affecting the acting participant p of the node n , this remediation strategy consists in changing the choreography so that the node n is removed from the Awareness Model. This case is equivalent to the removal of any participant-activated node discussed in Section 5.4.4, and it is therefore here omitted.

5.4.3.4 Remove the Throw Node

Given the Type 3 realizability defect on the node n_e affecting the acting participant p of n , this remediation strategy consists of removing the node n_e by deleting from the choreography its corresponding throw activity. While at first sight this remediation strategy appears relatively straightforward, the potentially large and unpredictable impact on the structure of the Awareness Model resulting from the deletion of the throw activity is such that we could not actually define invariants to avoid the introduction of other realizability defects as side-effect (see also Section 5.3). In particular, removing the throw activity entails removing the corresponding exception control-flow edges (see Section 3.4.1), which can lead to exception handlers becoming unreachable as well as and the introduction of new control flow edges between the previous activity in a block and the following one. Therefore, the empirically verification (see Section 5.2.2) of remediation plans produced by this strategy is always necessary.

5.4.4 Meta-Strategy: Delete the Affected Node

The remediation strategies proposed in Section 5.4.1 through Section 5.4.3 focus on particular types of realizability defects. There is, however, a remediation strategy that is applicable (within the boundaries set by the invariants, of course) to all types of realizability defects: the removal of the node affected by the realizability defect. All realizability defects are associated with the nodes on which they are diagnosed. Therefore, as long as the removal of the affected node does not cause the introduction of further realizability defects, removing it from the choreography is a legitimate way of disposing of the realizability defects associated with it.

The removal of a node is performed by using the “Remove Activity from Block” on the source activity of the node, and must therefore satisfy Invariant 7, Invariant 8, Invariant 13, Invariant 14 and Invariant 19. For atomic activities, namely message exchange and opaque activities, their removal sorts on the Awareness Model precisely the effect of removing the affected node. In case of composite activities, namely choice and iteration activities, their removal has also effect of removing the subgraphs generated by their branches and whatever activities are nested inside them. While the invariants are designed to handle the removal of composite activities, choreography modelers should be very conservative with the removal of potentially large chunks of their choreographies (think, for example, of the disruptive effect that would have on existing participant implementations). Additionally, as already discussed in Section 5.4.3.4, the case of removing a throw activity is very complicated because of its potentially massive implications on the structure of the Awareness Model and the reachability of its nodes. In this case, it is necessary to empirically verify the remediation plan.

Also important to point out is that the removal of an activity may violate the syntactical correctness of the choreography, in particular when the activity that is removed is the only one nested into a block (see also constraints applied to the “Remove Activity from Block” change operator, Section 5.1.3.1). To work around this issue, it is possible to add a skip activity to the block (using the “Insert Activity in a Block” change operator) before deleting the source activity of the affected node. Skip activities have void semantics (see also Section 3.3.2), so adding one in a block cannot introduce any realizability defect.

Besides adding skip activities, another possibility is to delete, instead of the activity, the block surrounding it. Deleting the block, however, generally brings further complications in terms of syntax violation: unless the block is a branch of a parallel or choice activity with three or more branches, deleting the block will cause a syntax error in its parent activity and is, therefore, not a generally applicable strategy.

5.5 On Dealing with Multiple Realizability Defects

In programming languages, design patterns and software development methodologies play fundamental roles in helping developers produce high-quality software. The same should apply to chore-

ography modeling. Currently, the state of the art of choreography modeling practices is mostly concerned on facilitating the transition from requirements to choreography models (see, e.g., [137, 19]). Little, however, seems to be known concerning how should modelers prioritize and order the application of remediation strategies over the process of evolving their choreographies to make them realizable. Indeed, to the best of this author’s knowledge, the only publication that specifically discusses how to combine multiple remediation strategies is [41], which limits itself to suggesting that one of the two types of the remediation strategies therein described should be applied before the other.

The goal of this section is to propose some guidelines based on this author’s experience with coding the prototype and empirically verifying the applicability of the remediation strategies presented in this chapter. The reader should be warned, however, that while they are consistent with the realizability analysis method proposed in Chapter 4 and with the software development experience of this author, these guidelines are presented without any claim of scientific validation or portability to other choreography modeling languages.

5.5.1 “Panta Rei:” Cure Earlier Realizability Defects First

The realizability analysis method proposed in Chapter 4 makes profuse use of Control Flow Graphs and control-flow algorithms to model the participant-awareness. Indeed, participant-awareness fundamentally *flows* from the initiating nodes towards the end nodes of the choreography. The flowing metaphor for participant-awareness allows us to formalize the following *partial order* for the remediation of multiple realizability defects afflicting one choreography:

1. Solve first realizability defects that are closer to the start node;
2. When a realizability defect affects a node inside the subgraph of a parallel activity, solve all the Type 3 realizability defects in that parallel activity first.

The rationale for the first rule is easy: fixing realizability defects may have side-effects on others. This implies that, as a rule of thumb, Type 2 realizability defects should be generally treated first, as initiating nodes are the “source” of participant-awareness. Then, on the basis of how the Type 2 realizability defects have been solved, solve Type 1 realizability defects “closest” to them, and so on.

The second rule, instead, sketches a partial order regarding the remediation of realizability defects inside parallel activities. As discussed in Section 5.4.3, the remediation of Type 3 realizability defects is limited by the fact that at most one participant can be immediately aware of any throw node. In practice, this tends to leave relatively few options to choreography modelers:

- Restructure the choreography to limit the scope the throwing and propagation of the exception in order to limit how many participant-activated nodes are interruptible by the throw node;
- Apply multiple times the Remediation Strategy 7 and have parts of the choreography where the only acting participant is the one immediately aware of the throw node.

The first option is not one that can generally be automated with a remediation strategy, because only the choreography modeler really knows what is the *intention* that motivates the modeling of exception throwing in the first place.

In the second case, modifying the choreography so that the only acting participant is the one immediately aware of the traversal of throw node imposes serious limitations on how to handle Type 1 and Type 2 defects affecting the interruptible nodes. For example, consider the case in which a Type 1 realizability defect affects the participant p a node n , n is interruptible by the throw node n_e , and n_e is affected by a Type 3 realizability defect because p is not immediately aware of n_e . In this case, the Remediation Strategy 2 would not be applicable to n , because it would introduce another Type 1 realizability defect on the awareness dependee of n . So, the only solution to solve the Type 1 realizability defect on n is applying the Remediation Strategy 1. However, applying the

Remediation Strategy 1 on n sorts exactly the same result as applying the Remediation Strategy 7. Therefore, taking care of all Type 3 realizability defects first seems to be a better option, as their remediation imposes strict limitations on how other realizability defects can be remediated.

5.5.2 Remediate Early, Remediate Often

Like most human ailments and virtually all software bugs and poor software design and architectural decisions, realizability defects seem to have a way to “fester” and get more expensive to fix in terms of time and impact on the overall choreography as time progresses and the choreography grows in size.⁵ As already pointed out in Section 5.5.1, participant-awareness flows along the paths of Awareness Models. Given this flow, the modeling of parts of a choreography that are “later,” farther from the start node is influenced by the parts of the choreography before it. Indeed, there are very often subtle dependencies between activities (e.g., “only participant p can send a message at this point”) that may affect how the choreography is modeled. Realizability defects that are left unsolved have the effect of altering these dependencies, misleading the choreography modelers during the further modeling.

Moreover, not only is the modeling of new parts of the choreography that is affected by the implicit dependencies among activities, but also the remediation of realizability defects. Remediation strategies have, for their very nature, a “ripple effect” in terms of participant-awareness. Fixing a realizability defect ultimately consists of modifications in the participant-awareness, which we already know to ripple through the choreography. It stands to reason that, due to the ripple effect of remediation plans, the cost of treating realizability defects later rather than sooner should increase in terms of both:

1. Time for the choreography modeler: solving the issue takes more and more complex remediation plans, and it takes longer to understand their implications on the logic of the choreography;
2. Side-effects on the logic of the choreography and the relative re-adjustments of the semantics of the roles, e.g., by adding or removing message exchanges.

Therefore, choreography modelers should strive to solve realizability defects as soon as they are detected and aim at modeling choreographies that are as much realizable as possible throughout the modeling process.

⁵This author has heard one too many times the sentence “Yeah, no problem, we can fix that Findbugs warning later” (cue in an ominous thunder). Such accursed statements seem to inescapably have the eldritch effect of preventing for all times the actual fixing of the issue. Because even when one eventually sits down and starts fixing the warnings, chances are that other parts of the codebase have grown subtly reliant on the flawed logic that causes the warning in the first place, and trying to fix it leads to the introduction of sudden, devastating bugs. Naturally, the good-minded developer gets unthankfully blamed on the grounds of: “It worked, you broke it.” Needless to say, this blame is predominantly originating from the developers who adamantly refused to fix the warning right away, thus adding insult to injury. (Any references to the worthy, almost heroic but albeit ill-fated efforts of one Andreas S. are absolutely intentional.)

Chapter 6

Implementation: The Crayons Prototype

This chapter introduces Crayons¹, a prototype that implements the techniques presented in this work. Crayons is an Eclipse-based IDE for ChorTex that integrates in the modeling process of choreographies the techniques for realizability analysis and remediation of realizability defects proposed in this work.

As shown in Figure 6.1, from the perspective of the user, Crayons adds to Eclipse three components:

1. A textual editor for ChorTex choreographies (Section 6.1);
2. A view that displays the Awareness Model of the ChorTex choreography been currently edited (Section 6.2);
3. A view that lists the realizability defects affecting the ChorTex choreography been currently edited and shows which remediation plans can be applied (Section 6.3).

Performance evaluation of the Crayons implementation in terms of realizability analysis is presented in Section 6.4

6.1 Editing ChorTex Choreographies

Due to the text-based nature of ChorTex, implementing the support for editing ChorTex choreographies in Eclipse was a very straightforward task thanks to Xtext³, an Eclipse-based framework for developing textual Domain-Specific Languages. Provided the BNF grammar presented in Section 3.2, Xtext generates all the code necessary for parsing and editing ChorTex choreographies. Moreover, Xtext also provides out-of-the-box validation of the syntax of DSL models based on the language's grammar. The validation can be extended with additional logic and quick fixes for validation errors can be implemented. In Crayons, both well-formedness verification and realizability analysis have been integrated as validation steps that Xtext performs on user typing and save actions. See, for example, Figure 6.2, which shows how realizability defects are marked in the

¹The name “Crayons” comes from the observation that, as often as not, business process as well as service choreographies are drawn by practitioners by hand on whiteboards (see, e.g., [106]). That is, rather uncomplicated tools such as markers and pencils are very widely used to model choreographies, at least in the first phases of the modeling process. Since the author entertains some very fond memories of painting with crayons a few decades past², the name pretty much picked itself.

²Said memories may or may not be shared with the same degree of fondness by various professional educators, architectural elements, furnitures of diverse size, make and stated purpose and, last but not least, my parents.

³Xtext website: <http://www.eclipse.org/Xtext/>

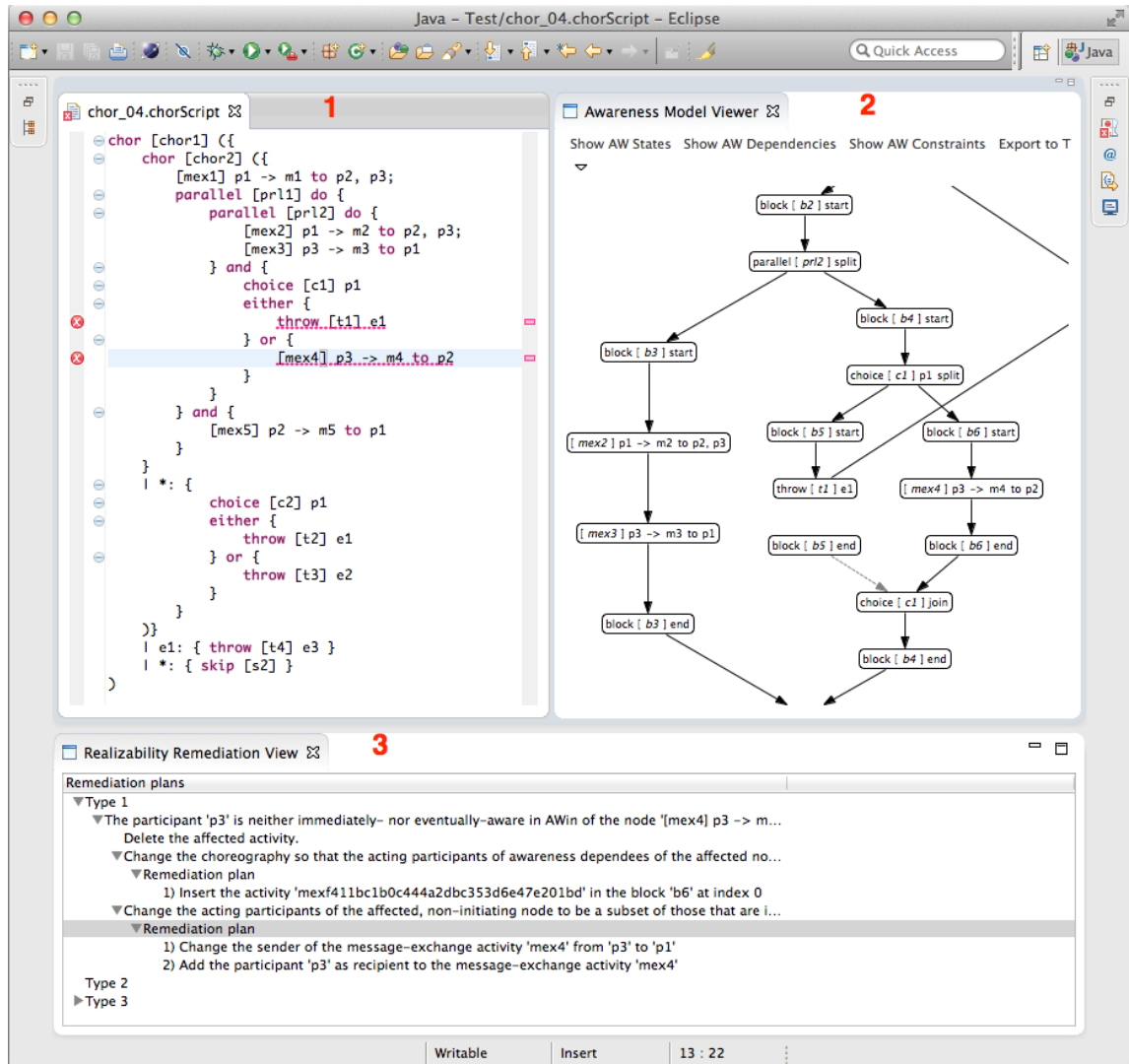


Figure 6.1: An overview of Crayons that highlights its three main components: (1) the ChorTex editor, (2) the AWM view and (3) the Realizability Remediation view.

ChorTex editor of Crayons: the source activities of AWM nodes affected by realizability defects are underlined with a red, dashed lines.

Quick fixes are implemented in Crayons to correct errors that violate well-formedness, e.g., renaming duplicated activity or message identifiers. Despite experimenting with it in earlier phases of prototyping, the remediation of realizability defects is currently not implemented as quick fixes. In Xtext, quick fixes are implemented in such a way that the only interaction with the user that should take place is selecting which quick fix to apply. Realizability defects and the associated remediation plans, however, require a lot of contextual information, which is provided in Crayons using the Awareness Model and Remediation Strategies views (see Section 6.2 and Section 6.3, respectively).

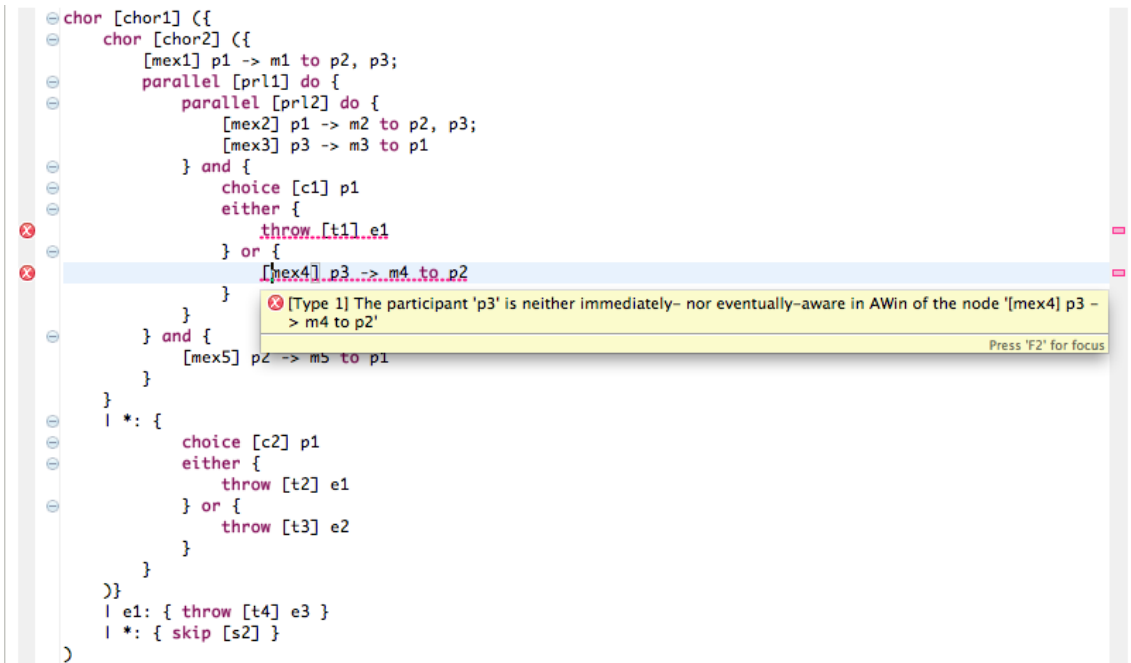


Figure 6.2: A Type 1 realizability defect highlighted as validation error in the ChorTex editor of Crayons.

6.2 Visualizing Awareness Models and Realizability Defects

The AWM view displays the AWM of the ChorTex choreography currently been edited. The AWM graph is rendered using the Zest graphical toolkit⁴. The following information can be toggled for display in the AWM view (see also Figure 6.3):

- Participant awareness;
- Awareness constraints, which are displayed in green or red depending on whether they are satisfied or not;
- Awareness dependencies (see Section 5.3.1).

Additionally, the AWM view also offers a simple “export to Tikz” features, that outputs the markup for the AWM in the Tikz toolkit for \LaTeX (a real life-saver for the examples presented throughout this work, truly).

6.3 Remediating Realizability Defects

The Realizability Remediation view is the hub for correcting realizability defects in the ChorTex choreography being edited (Figure 6.4).

The realizability defects affecting the choreography are broken down by type (Figure 6.4a). For each realizability defect, all the possible remediation strategies are listed, alongside with all remediation plans generate through them. Before selecting which remediation plan to apply, the user can evaluate how it would affect the choreography: with a right-click on the remediation plan, the user can open a comparison between the choreographies before and after applying the

⁴Zest website: <http://www.eclipse.org/gef/zest/>

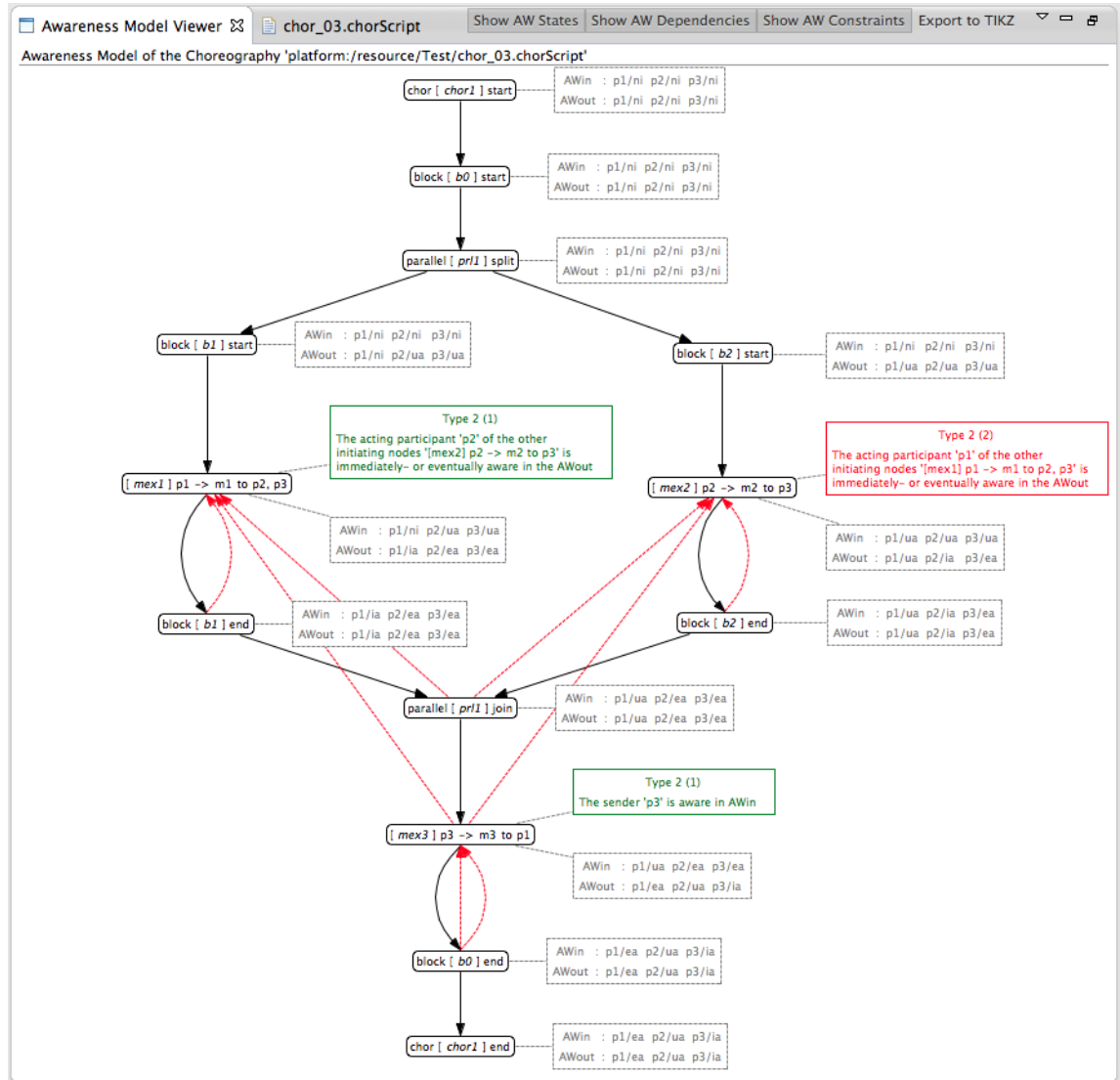


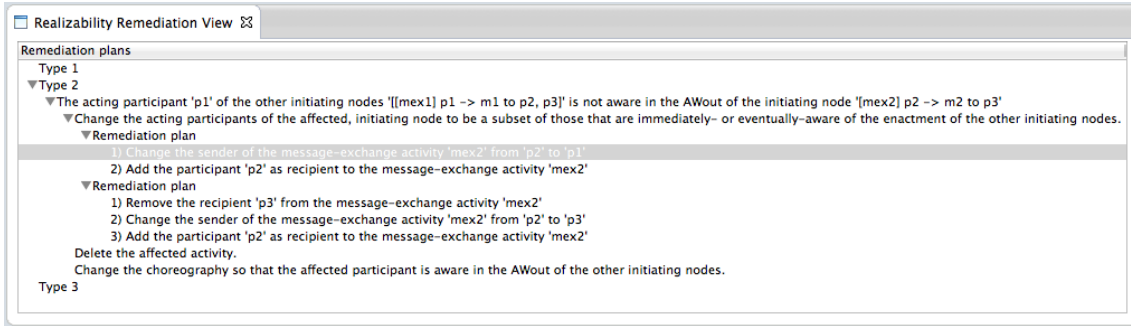
Figure 6.3: The AWM view.

remediation plan (Figure 6.4b). The comparison is realized using a customized Eclipse Comparison view, which should result extremely familiar to anybody who ever used versioning systems like Git or Subversion from within Eclipse.

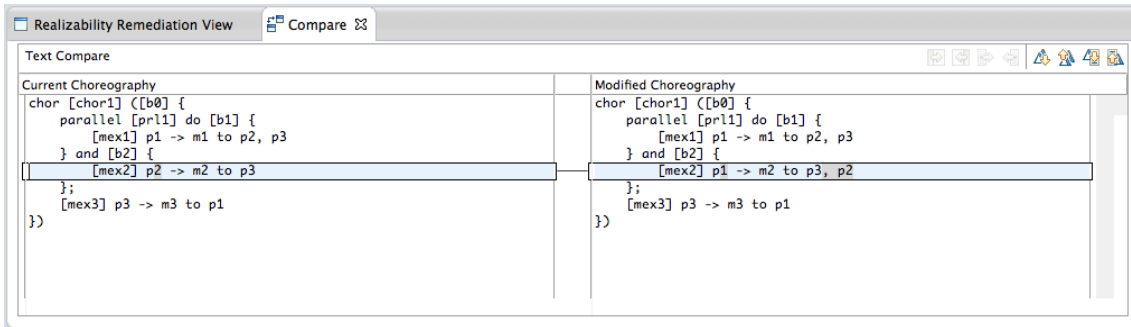
6.4 Performance Evaluation of the Realizability Analysis Method

In Section 4.7.1 is shown how the upper-bound complexity of the realizability analysis proposed in Chapter 4 is $\mathcal{O}(|\mathbb{N}(awm)|^3 \times |\mathbb{P}(chor)|)$, i.e., cubic with respect to the number of Awareness Model nodes generated from the choreography. Polynomial complexity is usually a promising sign that an algorithm is performant, but depending on the size of input and the polynomial factors, the execution time of an algorithm can anyways prove to be too large to be practical.

Table 6.1 presents the results of performing the realizability analysis 100 consecutive times on five choreographies of varying sizes in terms of both amount of activities and, therefore, Awareness



(a) Breakdown of the realizability defects affecting the choreography.



(b) Comparing a choreography before and after the application of a remediation plan.

Figure 6.4: The Realizability Remediation view.

Choreography	Activity count	Node count	Average analysis time over 1000 runs	Standard deviation
Chor 1	27	43	2.105 ms	0.771 ms
Chor 2	84	136	11.260 ms	2.286 ms
Chor 3	103	167	19.58 ms	12.914 ms
Chor 4	193	317	136.871 ms	73.217 ms
Chor 5	307	503	288.130 ms	117.648 ms

Table 6.1: Experiments with choreographies defining three participants and varying number of activities and nodes

```

1 chor ( {
2   chor ( {
3     parallel do { p2 -> m1 to p1 }
4     and {
5       p3 -> m2 to p1, p2;
6       opaque (p1, p2)
7     };
8     p3 -> m3 to p1, p2
9   }
10  | *: { skip } );
11 chor ( {
12   choice p1 either {
13     throw e1
14   } or {
15     p1 -> m4 to p2, p3
16   }
17 } )
18 }
19 | e1: { skip }
20 | *: { skip } )

```

Figure 6.5: Test choreography Chor 1

Model nodes. Unfortunately, as the best of this author’s knowledge there is no standard portfolio of choreographies available for benchmarking (and, even if there was one, differences in terms of expressiveness between choreography modeling languages would likely impact experimental results). The test choreographies have been selected as follows: Chor01 is the choreography displayed in Figure 6.5. Chor 2, Chor 3 and Chor 4 have been obtained copying the two choreographies nested in the body of Chor 1 and pasting them several times as additional activities inside Chor 1 body (of course, adjusting the message types as not to overlap). All test choreographies have three participants: despite the amount of participants appearing in the overall upper-bound complexity of the realizability analysis, in practice the amount of participant is irrelevant when compared to the importance of the amount of nodes.

The tests have been run as JUnit tests inside an instance of Eclipse on a MacBook Pro (late 2011) with a 2,2 GHz Intel Core i7 and 8 GB or 1333 MHz DDR3. To keep the tests as realistic as possible with normal usage scenarios, the machine has been used for an entire morning before running the tests, running browsers, L^AT_EX editors and even some large and resource-intensive video-games.⁵ Given the prototypical nature of Crayons, and thus the lack of any particularly optimization, the results are extremely promising: even with the extremely large Chor 4 and Chor 5, the realizability analysis runs in less than half a second, which is usually considered a short-enough time so that the average user will not be disturbed by the delay (see, e.g., [148]). It is also important to notice that the standard deviation in the cases of Chor 3, Chor 4 and Chor 5 is extremely high, roughly 50%. In other words, the fastest runs run in a fourth of the time of the slowest. The implementation in Crayons of the realizability analysis method is fully deterministic: the fixed-point algorithms always traverse the nodes of the Awareness Model in the same order, alphabetically sorted by identifier, which is generated by the source activity path in the choreography. Since our algorithms are deterministic, such a remarkably high standard deviation comes from the execution environment. All tests are run by the same Java Virtual Machine (JVM), which also runs the “surrounding” Eclipse instance. Subsequent runs of the realizability analysis do not share objects or resources, which means that new Java objects are created for every run (reusing parts of AWM object representations would be indeed the first optimization to be applied to Crayons). As a result, the heap space of the JVM is rather full, leading to frequent runs of the garbage collector, which block the execution thread of the realizability analysis and that, therefore, affect the overall performance. (The test environment is such that the JVM does not need to swap to virtual memory, which would further affect performances.) In the normal usage scenario of Crayons, the garbage collector is not triggered nearly as often and usually not during a run of the realizability analysis (the JVM would use idle cycles to run the garbage collector), leading to more consistent (and better) results.

⁵The best games of League of Legends are those one plays in the name of science!

Chapter 7

Conclusions & Future Work

Choreographies have great potential for facilitating the design and implementation of scalable, inter-organizational enterprise systems. However, the lack of central coordination, which is the defining characteristic of choreographies, comes with a steep price tag in terms of modeling pitfalls. In particular, choreographies should be *realizable*, i.e., their distributed message-based behavior should be such that the participants are *actually able to carry it out as intended*. Due to the lack of central coordination, it is surprisingly easy to specify choreographies that cannot be enacted as intended by their participants because of insufficient means to synchronize with one another. Such modeling issues that prevent choreographies from being enacted by their participants in part or in full are called *realizability defects*.

The state of the art is rich with realizability definitions and analysis methods for many different choreography modeling languages. Comparably little, however, has been done towards correcting the reliability defects that are found to affect choreographies. Realizability defects are extremely similar to issues that arise in parallel programming both in terms of nature and steep cost and complexity to detect and fix. Therefore, correcting realizability defects as early and as cheaply as possible is paramount.

This work aims at bridging the gap between the modeling process of choreographies and the correction of realizability issues. We tackle the case of choreographies specified with ChorTex, a choreography modeling language that we propose as a textual evolution of related work inspired by process algebras. The techniques proposed in this work have been designed to be seamlessly integrated in the Integrated Development Environments that are used throughout the modeling process. This work defines the concept of *participant awareness*, which consists of a symbolic representation of which events are visible to which participants during enactments of a choreography. Participant awareness is leveraged to the ends of realizability analysis by means of performant algorithms based on Control Flow Graph analysis that originate in the state of the art of programming languages and compilers. In this work is also put forward a comprehensive portfolio of remediation strategies that are tailored to solve all the various types of realizability defects affecting ChorTex choreographies. Heuristics are explored to define the order in which to deal with multiple realizability defects affecting one choreography. The realizability analysis and remediation strategies have been implemented in an Eclipse-based prototype that showcases our vision for integrating the correction of realizability defects with the modeling process of choreographies.

This concluding chapter is structured as follows. The research questions driving this work are revisited in light of the contributions proposed throughout it in Section 7.1. The single contributions are then discussed in terms of strengths and shortcomings in Section 7.2. Finally, future research directions are presented in Section 7.3.

7.1 Revisiting the Research Questions

In this section we revisit the research questions presented in Section 1.4 and correlate them to the contributions throughout this thesis.

What is the state of the art of modeling choreographies, realizability analysis and correction of realizability defects? Given the outlook provided in Chapter 2, the state of the art appears extensive and extremely rich with choreography modeling languages. However, the great majority of those languages are “academic,” meaning that they stem from the research community and lack the tooling and integration with middleware, enterprise systems and other modeling languages that is required for a successful adoption in the practice.

The landscape of “industrial” choreography modeling languages, namely those that have undergone a standardization process under the umbrella of organizations like the W3C or OASIS, seems to be affected by serious shortcomings. So far, two languages have been put forward that have had a chance at gathering the vendor support necessary for wide-spread adoption: WS-CDL and BPMN v2.0. Unfortunately, neither seem to have reached wide adoption. Like many other industrial modeling languages in the SOA and BPM landscape, both lack the formal foundation that would facilitate interoperability among implementations and enable solid, well-understood integration with other modeling languages.

Insofar realizability definitions and realizability analysis methods are concerned, the state of the art is also extremely rich. So rich and varied that there is actually a considerable fragmentation in terms of definitions of realizability and, consequently, realizability analysis methods. This fragmentation is to be fully expected, because a definition of realizability necessarily depends on the language it is specified on (see Section 2.3.1.2) and so many choreography modeling languages have been proposed. However, compared to the abundance found in state of the art of choreography modeling languages and realizability analysis, there is surprisingly little work concerning the correction of realizability defects. We posit that the gap in dealing with unrealizable choreographies, which ultimately motivated this work, is part of the reason why choreographies have failed to gain widespread adoption in the practice: without capable tools that support modelers in defining realizable, well-specified choreographies, the value proposition for practitioners is just not advantageous enough to warrant adoption.

What is an adequate definition of choreography realizability? As discussed in Section 2.1.2, choreographies are fundamentally technical contracts among the participants. As a technical contract, a choreography specifies the desired behavior to be displayed by the distributed systems constituted by the participant implementations that are based on that choreography. Therefore, an adequate definition of choreography realizability must ensure that all and only the desired messaging behaviors *can be executed to completion*. In other words, the choreography should be such that the resulting participant implementations can execute all and only the messaging behaviors (i.e., enactment traces, see Section 3.3) specified by the choreography and that no messaging behavior should deadlock because of oversights in the modeling of the choreography (such as when a choreography requires participants to perform behaviors that are impossible given the constraints on visibility of the message exchanges). These considerations are directly reflected in the definition of *strong realizability* adopted in this work (see Section 4.1).

It is important to notice that even stronger definitions of realizability may suit scenarios in which there are requirements not only in terms of the distributed messaging behavior to be exposed by the participant implementations, but also in terms of the perception of the single participant implementations concerning the status and completion of enactments. We sketch this avenue of future research in Section 7.3.

Given the chosen definition of realizability, how can it be verified in an effective manner? Given the definition of strong realizability adopted in this work, Chapter 4 presents

a method to verify that a ChorTex choreography is strongly realizable by means of fixed-point algorithms on Control Flow Graphs built from the choreography. Our method builds on top of the concept of awareness that is found in related work in the state of the art, and extends it to efficiently deal with asynchronous messaging. Logic predicates defined in terms of participant awareness, called *awareness constraints*, are evaluated on the Control Flow Graphs annotated with participant awareness, called Awareness Models, to ensure the strong realizability of the choreography. The resulting realizability analysis method has upper-bound complexity polynomial to the amount of activities that the choreography specifies (see Section 4.7.1). The performance measurements performed with the prototype are also extremely promising (see Section 6.4), and we believe that there are additionally great margins of improvement to be reaped by means of incremental realizability analysis (see Section 7.3).

Which remediation strategies can be defined to correct realizability defects? Given the definition of strong realizability adopted in this work, there are three different types of realizability defects that can affect ChorTex choreographies (see Section 4.5.1). The remediation strategies proposed in Section 5.4 tackle the three types of realizability defects. In a nutshell, the remediation strategies are specified in such a way that (see Section 5.2):

- produce remediation plans that are guarantee to solve at least one realizability defect;
- are guaranteed not to introduce further realizability defects.

The fact that the remediation strategies do not generate remediation plan that introduce further realizability defects is achieved by means of invariants that have been identified in Section 5.3. This framework for remediation strategies ensures that the choreography modelers are able to make their choreographies realizable in a finite amount of steps. Choreographies can be affected by any number of realizability defects at one time and the correction of one realizability defect may restrict or affect how others can be corrected. Therefore, in Section 5.4 we offer guidelines to deal with multiple realizability defects.

How can the theoretical results be leveraged to provide an integrated approach to solving realizability defects during the modeling process of choreographies? In Section 1.2 we have presented our vision for tools that seamlessly integrate the correction of realizability defects in the modeling process of choreographies in a way that is familiar to modelers with software development background and that has proven valid in the field of software development. Chapter 6 introduced Crayons, a prototype based on the Eclipse IDE that realizes our vision. Crayons leverages user interaction patterns (e.g., the comparison of the choreography before and after the application of a remediation plan) and user interface components that are extremely familiar to developers that use Eclipse for software development. Moreover, Crayons leverages the AWM concept proposed in Chapter 4 as a foundation of the realizability analysis to graphically present diagnostic information about the realizability defects to the modeler.

7.2 Revisiting the Contributions

This section revisits the various contributions provided in this thesis and discusses their strengths and shortcomings.

Categorization of Realizability Definitions

The categorization criteria for realizability definitions provided in Section 2.3.1 generalizes existing ones, in particular those provided by [79, 143], in the three following ways. First, the dimensions of realizability definitions are made applicable also to definitions that do not rely on equality relations between the behavior specified by the choreography and that of the composition of the peers.

Secondly, the communication model adopted by the participants can be specified with a fine-grained granularity by integrating the classification of communication models proposed in [132]. Thirdly, an entire new “axis” for classifying realizability definitions is proposed, which concerns how the realizability definition is termed with respect to the process of projecting the peers and comparing their composition to the choreography (does the definition require that a method to produce peers exist, or is it enough that peers are somehow given?) or, instead, if the realizability definition relies on other intrinsic properties of the choreography that, if verified, imply its realizability. This novel axis enables the categorization of realizability definitions and relative realizability analysis methods, such as the ones proposed in [100], that were previously not fitting the already-available classification criteria.

Besides offering a framework for creating an overview of the current state of the art concerning realizability definitions, our classification also works as a “check-list” of aspects that should be covered when establishing a new definition of realizability, which we hope it will help reducing the fragmentation currently affecting the state of the art.

Categorization of Realizability Analysis Methods

Naturally, realizability analysis methods are strongly related to the realizability definitions they verify. Based on our categorization criteria for realizability definitions, in Section 2.3.2 we have proposed a related categorization of realizability analysis methods, which we believe to be the first in the state of the art. Realizability analysis methods are categorized according to whether they are *constructive*, i.e., they rely on the process of projecting peers, composing them and comparing the composition with the choreography, or if they are *constraining*, i.e., they consist of verifying some intrinsic properties of the choreography, such as the constraints based on participants awareness that make up our realizability analysis. Of course there are also *mixed* methods that have both constructive and constraining aspects, but they tend to be fundamentally constructive methods compounded by the verification of some rather standard property like deadlock-freeness.

This novel classification of realizability analysis methods helps bringing order to the state of the art, in particular with respect to constructive and mixed methods. Section 2.3.3 consists of a survey that results from applying the proposed classification criteria to constructive and mixed methods. The most remarkable highlight of the survey is a rather widespread (and, frankly, alarming) underspecification of the communication model underpinning the choreographies, which considerably complicates the comparison of methods and the identification of research gaps. Another interesting observation that comes from our survey is the relatively little diversity in terms of peer projection methods that are adopted in constructive realizability analysis methods. Specifically, all the projection methods adopted in the surveyed works appear to be variations of the concept of *natural projection*, i.e., the transformation of the choreography into a peer by changing the message interactions and decisions as to reflect the perspective of one particular role, and removing all actions that do not involve that one role. Natural projection is very well suited when the modeling constructs used to model the choreography are similar to or have corresponding constructs for modeling peers; in case of considerable gap between peer and choreography modeling languages, however, other approaches should be applied, such as the one described in [142]. We believe the overwhelming reliance on natural projection to likely be the result of the scope of the articles that describe the realizability analysis methods. Put simply, all the articles we surveyed employ choreography modeling languages that can also be used, modulo minor variations, to model peers; due to the similarity between the choreography and peer modeling languages, there is simply no actual “pressure” to employ more refined projection methods. Additionally, the similarity between choreography and peer modeling languages simplifies not only the peer projection method, but also the evaluation of the behavioral similarity between choreography and peer composition. It would definitely be interesting to observe the kind of complications that arise when defining constructive realizability analysis methods using peer modeling languages that are very different from the choreography modeling ones.

The survey of constraining methods proposed in Section 2.3.4 was necessarily not as structured

as the one for constructive and mixed methods because constraintive methods tend to be extremely diverse and different from one another. In particular, the properties that are verified seem to depend very strongly from the choreography modeling language that is adopted as well as some fundamental underpinning concepts that are assumed, like awareness (see Section 2.3.4.1) or synchronizability (see Section 2.3.4.2). Therefore, directly comparing constraintive methods with one another often looks like comparing the proverbial apples and pears.

Categorization of Remediation Strategies

Similarly to the case of realizability definitions and realizability analysis methods, we also provide categorization criteria for remediation strategies (Section 2.4.2). Unlike the other classification criteria, however, these have been elicited from the very restricted state of the art (we could identify only four approaches) and are thus limited to describing fundamental properties of remediation strategies, namely which realizability definition they aim at enforcing, the extent of automation and if the changes they produce remove and whether they add or modify modeling elements (i.e., instances of modeling constructs) from the choreographies. In all likelihood, as the amount of related work grows, these classification criteria will have to be revised and extended to capture a finer granularity.

ChorTex

In Chapter 3 we introduced ChorTex, a choreography modeling language based on process algebras that uses a textual syntax reminiscent of natural language. ChorTex is based on Chor, another choreography modeling language proposed in [205]. Besides a completely revamped syntax, ChorTex replaces the *internal activity* construct offered by Chor, which we find highly controversial (in our opinion, internal behaviors of participant implementations are exclusive concern of the respective participants), with *opaque activities*, which we see as a “beach-head” for ad-hoc choreography modeling and dynamic, enactment-time behavior specification. We describe in Section 7.3 a few ideas to further refine the concept of opaque activities that would increase their usefulness and versatility.

Of course, ChorTex is no industrial language, so we have no expectation for it to be picked up by practitioners; its goal is to be a sandbox for investigating novel realizability analysis methods and remediation strategies. For reasons of brevity of this work, we decided to remove useful constructs like choreography referencing, as they would complicate immensely the presentation of the realizability analysis method. However, ChorTex is based on process algebras, which means that the fundamental constructs, as well as the techniques and methods we define throughout this work, are likely extremely easy to port to similar languages. Since ChorTex has never been meant for mass-adoption, we took the liberty of deliberately adding to it an exception throwing and handling mechanism that we consider nothing short of *toxic* for choreography modeling languages. As discussed in Section 4.7.2, the sudden activity termination caused by the propagation of exceptions and its limited visibility to participants require extremely careful handling at modeling time to prevent realizability defects. A similar construct is found, for example, in BPMN v2.0 in the guise of the termination end event. In light of our findings, it seems paramount for the research community to investigate alternative mechanisms for error detection and handling in choreographies that does not come with such potential pitfalls for modelers; future work in this direction is briefly sketched in Section 7.3.

Awareness-Based Realizability Analysis

In Section 4.2 is proposed the concept of *participant awareness*, which builds on top of the concept of awareness that has already been adopted in the state of the art (see Section 2.3.4.1), to fit to the asynchronous communication model we assumed for ChorTex (see Section 3.1). The Control Flow Graphs produced from ChorTex choreographies (see Section 3.4) are annotated with participant awareness using the Awareness Annotation Algorithm fixed-point algorithm (see Section 4.3). Logic

predicates, called *awareness constraints*, are evaluated on the resulting annotated Control Flow Graphs, called Awareness Models, to verify whether the choreography is strongly realizable or not. If the analyzed choreography is not strongly realizable, one or more of the awareness constraints is unsatisfied: each unsatisfied awareness constraint denotes a realizability defect.

Awareness Models are efficient to build and their size in terms of nodes and edges is comparable to the size of the choreography in terms of activities. This, together with the use of the parsing tree of ChorTex choreographies to detect parallel actions (see Section 4.4.2), gives to the proposed realizability analysis method a polynomial upper-bound complexity with respect to the amount of activities in the analyzed choreographies (see Section 4.7.1) and sub-second execution times on realistic usage environments also when analysing extremely large choreographies (see Section 6.4).

The relationship between the formulation of participant awareness as provided in this work and the constructs of ChorTex is discussed in Section 4.5.2. Additionally, we have put forward two conjectures related to participant awareness. The “Harmful unawareness” conjecture states that all realizability defects in a choreography (not necessarily specified with ChorTex) can be characterized in terms of insufficient participant awareness. Of course, this may require extensions or revisitations of the concept of participant awareness, which leads us to the second conjecture. The “Constructs imply participant awareness dimensions” conjecture, in fact, states that there is a direct relationship between the constructs specified by a choreography modeling language and the *dimensions* of participant awareness that are necessary to identify the realizability defects affecting choreographies modeled with that language. For instance, the modeling constructs of ChorTex are concerned exclusively with the order of actions performed by the participants, which is reflected by the formulation of participant awareness. However, were some constructs added to ChorTex that allow to describe time in a quantitative matter (e.g., timers and timeouts), then the notion of participant awareness would have to be expanded to encompass a *time* dimension.

Categorization of Realizability Defects for ChorTex Choreographies

Given the definition of strong realizability assumed in this work, three types of realizability defects have been found to affect ChorTex choreographies (see Section 4.5.1). Type 1 and Type 2 are similar, in that both are concerned to whether the acting participant of an action, such as the participant that has to dispatch a message, is aware that it can perform that action without violating the choreography. The difference between Type 1 and Type 2 is that the latter deal with actions that may initiate an enactment and the former with all other actions. Type 3 are the “other flip of the coin” with respect to Type 1 and Type 2. Type 3 realizability defects consist of participant that do not know if they *cannot* perform some actions without risking violating the choreography. Type 3 realizability defects are a direct consequence of the exception handling mechanism of ChorTex, and in particular of the fact that the propagation of an exception may immediately interrupt activities that are been executed in parallel. The actions specified by those interrupted activities cannot therefore be executed without violating the choreography, and this requires extreme care when modeling a choreography, as well as strongly limiting the use of exception handling in choreographies that need to be strongly realizable (see Section 4.7.2).

Change Algebra for ChorTex Choreographies

The change algebra for ChorTex choreographies defined in Section 5.1 is designed to be *minimal* and *complete*. Minimal means in this context that each change operator modifies the least amount of information to achieve its goal and no two change operators modify the same type of information in the same way. The change algebra is complete because every aspect of a ChorTex choreography can be modified through the provided change operators. Moreover, the change operators are designed as to preserve well-formedness, which is a necessary condition for strong realizability of ChorTex choreographies. Since the change operators are minimal, changes that may be perceived as “atomic,” such as inverting the order of two activities, actually require the sequential application of multiple change operators (see Section 5.1.4).

Invariants to Avoid Introducing Realizability Defects

Section 5.3 presents an analysis of how the application of change operators can result in the introduction of new realizability defects. Invariants are elicited that ensure that the application of change operators does not result in new realizability defects. The invariants are all defined on the basis of the current choreography (or, more specifically, of the Awareness Model resulting from it) and the operational semantics of the change operator.

Invariants have not been defined for all change operators, but only for those actually used in our remediation strategies. The change operators that have not been examined fall in either of the following two categories: *irrelevant* or *disruptive*. Irrelevant change operators, like the one that renames an activity, cannot be used to define remediation strategies because they do not modify the distributed messaging behavior specified by the choreography. Disruptive change operators, on the other hand, change the distributed messaging behavior *too much* to actually be able to forecast their impact on the choreography without applying them. This is the case for most change operators that result in complex modifications to the structure of the Awareness Models, e.g., those that change how exceptions propagate and which activities are interrupted. In this latter case, when necessary in the scope of remediation strategies, we rely on *empirical validation*: to evaluate the impact of a change operator on a choreography, we compare the outcome of the realizability analysis “before and after,” i.e., on the current choreography and on the one resulting from the application of the change.

Remediation Strategies for ChorTex Choreographies

Our library of remediation strategies for dealing with realizability defects in ChorTex choreographies is presented in Section 5.4. The remediation strategies are defined so that they produce remediation plans that are (1) *useful*, i.e., that do solve at least one realizability defect, and (2) *not harmful*, i.e., the produced remediation plans do not introduce further realizability defect (see Section 5.2.1). Several remediation strategies have been identified to deal with each of the types of realizability defects that can affect ChorTex choreographies. Moreover, the generic remediation strategy of removing the node affected by the realizability defect has also been investigated.

The library of remediation strategies described in this work is believed to be *complete*, meaning that one remediation plan can always be generated to solve a given realizability defect. In the case of choreographies affected by more than one realizability defect, the interplay of Type 3 realizability defects with the other two types may prevent any remediation plan from being automatically generated if the realizability defects are tackled in the wrong order (see following paragraph).

Guidelines to Deal with Multiple Realizability Defects

As implied by the previous paragraph, the order in which a modeler undertakes the correction of multiple realizability defects has import on whether it is possible to automatically generate remediation plans with our remediation strategies or not. This is normal considering that participant awareness is calculated with a fixed-point algorithm applied to annotated Control Flow Graphs. In fact, participant awareness fundamentally *flows* among nodes in the Control Flow Graphs following the control flows from source to target. Therefore, modifications to the participant awareness of nodes “upstream” (sources of control flows) end up affecting those “downstream” (the targets of those control flows).

In Section 5.5.1 we provide heuristics to determine the order in which realizability defects should be solved. Due to the flow effect of participant awareness, correcting realizability defects that affect upstream nodes should be prioritized by modelers over correcting the realizability defects downstream (see Section 5.5.1). Exceptions to this general rule are Type 3 realizability defects. Since Type 3 realizability defects involve entire areas of the choreography, i.e., those parts that can be executed in parallel, solving them tends to have repercussions on the various parallel branches. Thus, solving the realizability defects on those parallel branches should be postponed until the Type 3 realizability defects have been dealt with. Another aspect of how to deal with realizability

defects during modeling is treated in Section 5.5.2. Due to similarities with the practice and theory of software engineering and development, we make the case that modelers should strive to correct realizability defects as soon as possible during the modeling process.

The Crayons Prototype

Chapter 6 presents the Crayons prototype, based on the Eclipse Integrated Development Environment, that implements a modeling environment for ChorTex deeply integrated with an implementation of the realizability analysis method and remediation strategies presented in this work.

7.3 Future Work

In this section we outline future research directions to follow up on this work.

Improvements for ChorTex

As already mentioned, there are constructs that are considered useful for choreography modeling, but did not make it into ChorTex for reasons of scope of this work. As discussed in Section 4.5.2, the constructs of a choreography modeling language determine the types of realizability defects that can affect choreographies. Therefore, adding new or modifying existing constructs of ChorTex may require adaptations to the realizability analysis and remediation strategies, which we will endeavor to sketch out in broad strokes in the remainder.

Choreography referencing: ChorTex supports choreography nesting, but not referencing (see 3.6.2). That is, it is not possible to “reuse” one choreography snippet several times in one or multiple choreographies. While extremely likely not affecting directly the types of realizability defects that may appear in ChorTex choreographies, choreography referencing requires changing the realizability analysis method to use inter-procedural control flow analysis [160]. Additionally, allowing the reuse of snippets means that, in terms of remediation of realizability defects, it might be undesirable to modify the snippets (which may be reuse elsewhere, potentially introducing issues), which would introduce another dimension for deciding for one remediation plan over another.

Channel passing: Many process algebras have the concept of channel passing, which allows to “invite” at runtime participants to take part in the enactments. Adding channel passing to ChorTex would introduce new types of realizability defects:

1. Acting peers that have not yet been invited cannot perform actions (which may result in deadlocks [66]);
2. Sender does not know yet the identity of recipient that have been invited by some other participant (see also [44]);
3. Conflicting invitations to multiple participants for playing the same role.

The analysis of which participant knows the identity of which other, as well as how the information about invited participants is propagated across the enactments, should be rather straightforward to perform based on reaching-definitions algorithms (inviting a participant to play a role counts as an “assignment” to that role’s identifier). The remediation strategies, on the other hand, are unlikely to be equally straightforward, in particular those that will deal with conflicting invitations.

Refining the concept of opaque activities: In Section 3.2 we have characterized opaque activities as yet-unspecified parts of the choreographies. In order to be able to evaluate the realizability of a choreography with opaque activities, we have required that all acting participants of an opaque activity are eventually-aware of the beginning and completion of its execution (see Section 4.4.1). All participants eventually-aware of the beginning and completion of an activity, however, does not fit the use-case of open choreographies, i.e., choreographies that are specified ad-hoc at runtime by their participants. An interesting option would be make integral part of the specification of opaque activities which participants need to be immediately or eventually-aware of their beginning (which are likely to be the initiators of those opaque activities) and which participants will be immediately or eventually-aware at their end. In terms of changes to realizability analysis and remediation strategies, allowing for these revamped opaque activities would require a few, rather straightforward changes. Besides making opaque activities more flexible, this approach would have the effect of making participant-awareness integral part of the syntax of ChorTex. Making participant-awareness a first-class citizen in the specification and modeling of choreographies seems a natural evolution of the work presented in this thesis.

Data-based decisions: ChorTex choreographies do not have a “data dimension.” The interactions between participants are characterized exclusively by the type identifier of the exchanges messages. Allowing decisions to be performed on the content of messages would introduce a new type of realizability defect (which could be called “history sensitivity,” see also [42]) in which the decision maker does not have access to all the latest messages that delivered the data involved in the decision.

Impact Assessment of Remediation Plans

In general, not all remediations plans should be equally desirable to designers. Future research directions should investigate what makes one remediation plan more desirable than another. Indeed, there are several aspects that one could take into account when evaluating which remediation plan to apply, such as functional aspects like whether adding some interaction is compatible with the business scenario captured by the choreography and non functional aspects like Quality of Service (e.g., length of enactments, bandwidth consumed, impact of network degradation) and security (confidentiality of exchanged messages, non-repudiation, etc.).

Investigating Realizability with Clear Termination

In this work we assume a rather traditional definition of realizability which requires deadlock-freeness and language equivalence in terms of conversations between the choreography and the composition of projected participants. However, perhaps surprisingly, strong realizability may not always be *strong enough*. From the point of view of the conversations that will be enacted by participant implementations, strong realizability is perfectly satisfactory. What may prove useful in some scenarios, however, are strong guarantees that the participants can observe when their role in enactments is over. These type of guarantees is known as *clear termination* [119], i.e., as the property of a choreography that “a partner always can compute, whether he will be sender or receiver of any messages in the sequel.” Rephrasing the definition of clear termination provided in [119] according to the terminology adopted in this work, we have the following:

Definition 7.1 (Clear Termination). A choreography presents the *clear termination* property if, at any point in time of any valid enactment of that choreography, every participant can always compute if it will be involved as either sender or receiver of any message exchange in the remainder of the enactment.

Clear termination is a desirable property of choreographies because of the way it affects the participant implementations. Without clear termination, some participant implementations will be unable to decide if their part in an enactment has ended, or even if the enactment itself has

been completed. From the point of view of their participants, those participant implementations are *hanging*, consuming computing resources that cannot be freed.

The lack of clear termination can also give rise to security concerns [107]. If a participant implementation is waiting for more messages to come, a malicious attacker could spoof the identity of another participant and “hijack” the conversation long after the enactment has completed. Whether such a security flaw exists and it can be exploited depends on the technologies used to implement the participant implementations and, in particular, the aspects related to confidentiality, authenticity and non-repudiation. However, clear termination of choreographies is likely to reduce the “window of opportunity” for such malicious behaviors.

Monitoring and associated activities are also highly impacted by the absence of clear termination. Offline monitoring, i.e., the analysis and processing of the logs of completed interaction (see Section 2.1.2), is unavailable to those participant implementations that are unable to know if their part in an enactment is over. Offline monitoring is involved in many aspects of BPM such as process analysis and process mining (see, e.g., [185]) as well as compliance management and auditing. For inter organizational processes that require auditing, the lack of clear termination is likely to be a “deal-breaker.”

Extending the contributions of this work to address clear termination would give rise to another type of realizability defect, which would occur for participants that, at some point in an enactment, cannot know if they will further take part in it or not. Some early research results lead us to believe that clear-termination analysis can be superimposed on the realizability method proposed in this work using a variation of a reaching definitions algorithm and another dimension of participant awareness. This additional dimension of participant awareness would describe whether, from a given node and for an arbitrary end node, the participant is either certain to be involved in actions (as acting participant or message recipient) or certain that, if the enactment ends with that end node, it will not be involved in any more actions. If neither of the previous cases is true, there is a realizability defect denoting lack of clear termination for that participant.

Incremental Realizability Analysis

In the performance evaluation of the prototype, we mentioned that subsequent runs of the realizability analysis do not share data-structures, i.e., each analysis is conducted “from scratch” (see Section 6.4). Granted, in the implementation one could actually reuse in large part existing objects, like the in-memory representation of Awareness Models, but dealing with incremental realizability analysis is far from being only an implementation question. Indeed, one could characterize the delta in terms of Awareness Models snippets that results from a change (which, to some extent, we have already done when discussing awareness dependencies, see Section 5.3.1) to calculate how the snippet affects inbound and outgoing participant awareness. This would avoid recalculating the entire participant-awareness fixed point for every change, drastically reducing the computational resources needed to perform the realizability analysis. (In fact, the upper-bound complexity is cubic with respect to the number of nodes, and that is mostly due to the fact that, in the worst case, one has to perform n^2 iterations of the fixed-point algorithm to reach a fixed point, see Section 4.7.1). A deeper understanding of how change operators affect participant awareness would also enable the definition of further, more complex remediation strategies, which may result in fewer steps to be performed by a modeler to correct all realizability defects.

Modeling Practices and Anti-Practices for Choreographies

In the scope of programming languages there is a wealth of knowledge available in terms of best coding practices and anti-practices. Similar knowledge, unfortunately, is not currently available for choreography modeling. In the industry there are portfolios, like the RosettaNet standard¹, of abstract business processes that modelers can use as “inspiration” or canvas to model choreographies and executable orchestrations. However, this does little to support modelers in writing “good”

¹RosettaNet consortium website: <http://www.rosettanet.org/>

choreographies that technically use properly and correctly the modeling constructs provided by the languages and that are easy to understand, to implement and to maintain.

In this work we started “scratching the surface” insofar how “bad” constructs affect the specification of realizable choreographies. In particular, we have discussed the pitfalls and shortcoming of termination constructs like the exception handling mechanism of ChorTex. In our view, this proves that, in the same way “bad” programming constructs lead to worse code, bad choreography modeling constructs lead to worse choreographies. One avenue that could be explored by the research community would be to tap into exception handling mechanism defined in the scope of other fields, like long-running database transactions and error detection and handling in networking protocols to define better-behaved termination constructs for choreographies.

Appendix A

Proof of Correctness of the Awareness-based Realizability Analysis

This section proves that the awareness constraints presented in Section 4.4 and the assumption of well-behavedness of peers are sufficient and necessary conditions for the strong realizability of ChorTex choreographies as defined in Section 4.1. This proof follows loosely the structure of the one provided in [52] with respect to strong realizability of collaboration diagrams, and it is divided in three steps:

1. Build peers that can execute the roles of participants in ChorTex choreographies (Section A.1);
2. Show that the satisfaction of the awareness constraints and the well-behavedness of participants are sufficient conditions to ensure strong realizability of ChorTex choreographies (Section A.2); this step is further divided in proving that:
 - (a) The composition of the peers is stuck-free (Section A.2.1);
 - (b) The composition of the peers can execute all and only the conversations specified by the choreography (Section A.2.2).
3. Show that the satisfaction of the awareness constraints and the well-behavedness of participants are necessary conditions to ensure strong realizability of ChorTex choreographies (Section A.3).

A.1 Building Peers

Peers for the participants in a ChorTex choreography chor can be modeled as deterministic FSMs, each one representing how one participant executes its role in the choreography by playing out the operational semantics rules of ChorTex on the basis of its knowledge of the status of the enactment.

The peer \hat{p}_i for the participant p_i in the set $\mathbb{P}(\text{chor})$ of participants of chor is built as the deterministic finite state machine $(S, s_{\text{init}} \in S, T \subset S \times \langle r_1, \dots, r_n \rangle_{r_i \in R} \times S)$, where S is the set of states, s_{init} is the initial state and T is the set of transitions between states in S through the application of a sequence of the operational semantics rules R of ChorTex defined in Section 3.3.2.

The states of the peers are defined in terms of states of the activities of the choreography. Given the set of activities $\mathbb{A}(\text{chor}) := \{a_1, \dots, a_n\}$, the set of possible states for the activity \mathbf{a} in enactments of a choreography is denoted by as $\Delta^{\mathbf{a}}$, see Section 3.3.2. Recall that the notation

$\delta[\mathbf{a} \setminus \mathbb{S}]$ means that the activity \mathbf{a} is in the status \mathbb{S} . The set S of all possible states for a peer \hat{p}_i in the choreography \mathbf{chor} is defined as follows:

$$S := (\Delta^{a_1} \cup \{?\}) \times \dots \times (\Delta^{a_m} \cup \{?\})$$

The state space S of a peer is necessarily finite: each choreography contains a finite number of activities and the state of each activity can assume only a finite amount of values (see Section 3.3.2). In practice, the actual state space of a peer is necessarily a subset of the combination of all possible activity states, because there are many combinations that are not compatible with the operational semantics of ChorTex. For example, consider the following state, in which $\mathbb{A}(\mathbf{chor}) := \{\mathbf{chor}, \mathbf{a}_1, \dots, \mathbf{a}_n\}$:

$$s := (\delta[\mathbf{chor} \setminus \mathbb{S}], \delta[\mathbf{a}_1 \setminus \mathbb{S}], \dots, \delta[\mathbf{a}_n \setminus \mathbb{S}])$$

The peer state above is one in which the root choreography \mathbf{chor} is completed, but all the activities nested into it are in \mathbb{S} , i.e., their initial state; given the operational semantics rules of ChorTex, this state is never achievable.

The initial state s_{init} of each peer in a choreography \mathbf{chor} with activities $\{\mathbf{chor}, a_1, \dots, a_n\}$ is the following:

$$s_{\text{init}} := (\delta[\mathbf{chor} \setminus \mathbb{S}], \delta[a_1 \setminus \square], \dots, \delta[a_n \setminus \square])$$

The symbol \square denotes the blank state of an activity, i.e., the state before the activity gets initiated for the first time.

The states of peers are connected with transitions that represent the execution of sequences of the operational semantics rules presented in Section 3.3.2. Each transition represents one or more operational semantics rules being executed in sequence. Each sequence of operational semantics rules is *maximal*, i.e., it is composed of all the rules that can be executed one after the other given the source state of the transition. All transitions, except for those that connect the initial state with others, are triggered by an action, such as the dispatch of a message exchange. Indeed, all maximal sequences of operational semantics rules terminate with either the completion (either successful completion or termination) of the choreography, or the wait for the next action to occur. In this respect, it is helpful to imagine the maximal sequences of operational semantics rules as the programming logic of a simple command-line shell: after an input (observing or taking an action) the command is executed until its completion (the beginning of the wait for the next action). The transitions that connect the initial state with others do not have trigger actions. Given the initial state, there is always at least one rule that can be executed without observing an action, namely the Choreography Initiation rule, which initializes the body of the choreography after the choreography itself has been initialized. Given the fact that the root choreography is initialized in the initial state, the Choreography Initiation rule is always the first to be applied in any enactment.

Lemma (Peer Determinism). The peers constructed for ChorTex participants are deterministic.

Proof. This lemma is a direct consequence of the unambiguous way that operational semantics rules are defined in Section 3.3.2 plus the Well-formedness requirement 4. In fact, the participants cannot mistake events they observe (“Unequivocal message exchanges” design assumption: given a message they observe being dispatched, there is exactly one activity that can generate it). Moreover, given one enactment status, there can be only one maximal sequence of operational semantics rules applicable to it (unambiguous operational semantics). Therefore, there cannot be two transitions triggered from one state by the same event, and thus the peers are deterministic. \square

A.2 Sufficiency of Conditions for Strong Realizability

The goal of this section is to prove that well-behavedness of participants and the satisfaction of the awareness constraints are sufficient conditions for the strong realizability of ChorTex choreographies.

A.2.1 Stuckness Freedom of Composition of the Peers

The composition of peers built as specified in Section A.1 from a strongly realizable choreography is stuck-free if, after any action is performed, either the enactment is completed or at least one peer is able to perform a transition to another state.

Proof. The proof of stuck-freeness of peer compositions created from strongly realizable ChorTex choreographies is based on the following observation on enactment states. Given an enactment state, there are the following two possibilities:

1. There are no more actions to be performed, and hence the enactment state is in a final state and the enactment is over;
2. There are one or more actions that can be performed without violating the choreography.

The first case already satisfies the definition of stuck-freeness.

In the second case, the actions a_1, \dots, a_n can be performed by some peers without violating the choreography. To prove stuck-freeness, we need to prove that at least one peer is able to perform a transition. Performing a transition means executing a sequence of operational semantics rules that change the state of the peer. The state of the peer is defined by its perception of the states of the single activities of the choreography. Therefore, a transition that causes a change of state of a peer also causes a change of state of one or more choreography activities. Since the choreography from which the peer composition has been created is strongly realizable, all its awareness constraints are satisfied. Specifically, all its Type 1 and Type 2 awareness constraints are satisfied. In turn, this guarantees that the acting participants of each of the a_1, \dots, a_n actions are aware that they can perform those actions. Since performing an action always change the state of at least one activity, all the peers of those participants can perform transitions from their current states. \square

A.2.2 Language Equivalence between Strongly Realizable Choreography and Composition of its Peers

A peer composition is language equivalent to a choreography if the former can execute without violations all and only the enactments specified by the latter.

Proof. The fact that all enactments specified by the choreography can be executed by a composition of well-behaving peers is straightforwardly given by how the peers are constructed (see Section A.1). We prove by contradiction that only the enactments specified by a strongly realizable choreography can be executed without violations by the composition of its well-behaving peers. Assume that the composition of the peers can perform an enactment trace that violates the choreography. Since the choreography is strongly realizable, all its awareness constraints are satisfied. Therefore, in any enactment state all acting participants of any action are aware of the fact that performing their action would violate the choreography. The only way a choreography can be violated is by having a peer perform an action that is not allowed in the current enactment state. The peers that perform the action that violates the choreography must therefore know that their action would cause the violation of the choreography. Since the peers are assumed to be well-behaved, this causes a contradiction, namely that well-behaved peers knowingly violate the choreography (see Definition 4.2). \square

A.3 Necessity of Conditions for Strong Realizability

The goal of this section is to prove that both well-behavedness of participants and the satisfaction of the awareness constraints are necessary conditions for the strong realizability of ChorTex choreographies.

A.3.1 Necessity of Well-Behavedness of Participants

Well-behavedness is straightforward to prove as a necessary condition for the strong realizability of choreographies. A non well-behaved participant can be classified as *malicious*: it is a participant that deliberately or accidentally (e.g., because of software bugs in the peer implementations) violates the choreographies it takes part to. Imagine a malicious participant that, as soon as it is made aware of the existence of an enactment (e.g., by receiving the first message directed to it) immediately violates the choreography by sending a message that is not specified in the choreography to one or more other participants. Not necessarily all enactments in a choreography involve all participants, so it might be the case that the composition of the participant implementations could actually perform correctly some of the enactments. However, some of the enactments necessarily include the malicious participant (if not, it would not be a participant to begin with). Therefore, since only some of the enactments of the choreography can be correctly executed by the composition, the latter is not language equivalent with the choreography, and therefore the choreography is not strongly realizable.

A.3.2 Necessity of Awareness Constraints Satisfaction

The goal of this section is to prove the necessity of the satisfaction of all awareness constraints for the strong realizability of a choreography. We prove it by contraposition, showing that the assumption that some awareness constraints are not satisfied on an arbitrary choreography implies that the choreography is not strongly realizable.

This proof is already implicitly provided in Section 4.4, in which are analyzed the three possible types of realizability defect and how to verify for their absence in a choreography. We make it now explicit, considering the case of arbitrary choreographies in which one awareness constraint per type is not verified, and argue that such a choreography cannot be strongly realizable.

The cases of Type 1 and Type 2 are identical, as they both relate to stuck-freeness: if a Type 1 or Type 2 awareness constraint is not verified, then one acting participant of an action is not aware of when that action can be enacted without violating the choreography. Since the participant is assumed to be well-behaved, it means that its action will never be executed, hence all enactments in which that action has to be performed will eventually get stuck. Since the choreography is not stuck-free, it is also not strongly realizable.

The case of Type 3 awareness constraints is related with language equivalence. Specifically, if there is one unsatisfied Type 3 awareness constraint, then the composition of the peers is necessarily able to perform some enactment that is not specified by (i.e., violates) the choreography. Straightforwardly, these violating enactments all foresee the execution of one or more actions that have been rendered non-enactable by the throwing of the exception related to the unsatisfied Type 3 awareness constraint, see for instance the example shown in Figure 4.9. Since the peer composition can execute some enactments that violate the choreography, the former is not language equivalent with the latter, and therefore the choreography is not strongly realizable.

Bibliography

- [1] *2009 IEEE International Conference on Services Computing (SCC 2009), 21-25 September 2009, Bangalore, India*. IEEE Computer Society, 2009.
- [2] Wil M. P. van der Aalst et al. “Multiparty Contracts: Agreeing and Implementing Interorganizational Processes”. In: *Comput. J.* 53.1 (2010), pp. 90–106.
- [3] Martín Abadi, Leslie Lamport, and Pierre Wolper. “Realizable and Unrealizable Specifications of Reactive Systems”. In: *ICALP*. Ed. by Giorgio Ausiello, Mariangiola Dezani-Ciancaglini, and Simona Ronchi Della Rocca. Vol. 372. Lecture Notes in Computer Science. Springer, 1989, pp. 1–17. ISBN: 3-540-51371-X.
- [4] Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.
- [5] Michael Adams et al. “Worklets: A Service-Oriented Implementation of Dynamic Flexibility in Workflows”. In: *OTM Conferences (1)*. Ed. by Robert Meersman and Zahir Tari. Vol. 4275. Lecture Notes in Computer Science. Springer, 2006, pp. 291–308. ISBN: 3-540-48287-3.
- [6] A.V. Aho. *Compilers: principles, techniques, & tools*. Addison-Wesley series in computer science. Pearson/Addison Wesley, 2007. ISBN: 9780321486813. URL: <http://books.google.com/books?id=n3eVQgAACAAJ>.
- [7] Marco Alberti et al. “Computational Logic for Run-Time Verification of Web Services Choreographies: Exploiting the *SOCS-SI* Tool”. In: *WS-FM*. Ed. by Mario Bravetti, Manuel Núñez, and Gianluigi Zavattaro. Vol. 4184. Lecture Notes in Computer Science. Springer, 2006, pp. 58–72. ISBN: 3-540-38862-1.
- [8] Marco Alberti et al. “Verifiable agent interaction in abductive logic programming: The SCIFF framework”. In: *ACM Trans. Comput. Log.* 9.4 (2008).
- [9] Alessandro Aldini and Marco Bernardo. “On the usability of process algebra: An architectural view”. In: *Theor. Comput. Sci.* 335.2-3 (2005), pp. 281–329.
- [10] Midhat Ali, Guglielmo De Angelis, and Andrea Polini. “ServicePot - An Extensible Registry for Choreography Governance”. In: *SOSE*. IEEE Computer Society, 2013, pp. 113–124. ISBN: 978-1-4673-5659-6.
- [11] Frances E. Allen. “Control flow analysis”. In: *SIGPLAN Not.* 5 (7 July 1970), pp. 1–19. ISSN: 0362-1340. DOI: <http://doi.acm.org/10.1145/390013.808479>. URL: <http://doi.acm.org/10.1145/390013.808479>.
- [12] Gustavo Alonso, Peter Dadam, and Michael Rosemann, eds. *Business Process Management, 5th International Conference, BPM 2007, Brisbane, Australia, September 24-28, 2007, Proceedings*. Vol. 4714. Lecture Notes in Computer Science. Springer, 2007. ISBN: 978-3-540-75182-3.
- [13] Rajeev Alur, Kousha Etessami, and Mihalis Yannakakis. “Realizability and verification of MSC graphs”. In: *Theor. Comput. Sci.* 331.1 (2005), pp. 97–114.

- [14] Vasilios Andrikopolous. “A theory and model for the evolution of software services”. PhD thesis. Tilburg University, Oct. 2010.
- [15] Farhad Arbab and Marjan Sirjani, eds. *International Symposium on Fundamentals of Software Engineering, International Symposium, FSEN 2007, Tehran, Iran, April 17-19, 2007, Proceedings*. Vol. 4767. Lecture Notes in Computer Science. Springer, 2007. ISBN: 978-3-540-75697-2.
- [16] Liliana Ardissono et al. “PERCHE: A Public Registry for Choreographies”. In: *System and Information Sciences Notes 1.4* (2007), pp. 353–358.
- [17] Assaf Arkin et al. *Web Service Choreography Interface (WSCI) Version 1.0*. W3C Note. Aug. 2002. URL: <http://www.w3.org/TR/wsci>.
- [18] Giorgio Ausiello, Mariangiola Dezani-Ciancaglini, and Simona Ronchi Della Rocca, eds. *Automata, Languages and Programming, 16th International Colloquium, ICALP89, Stresa, Italy, July 11-15, 1989, Proceedings*. Vol. 372. Lecture Notes in Computer Science. Springer, 1989. ISBN: 3-540-51371-X.
- [19] Marco Autili et al. “A development process for requirements based service choreography”. In: *RESS*. IEEE, 2011, pp. 59–62. ISBN: 978-1-4577-0939-5.
- [20] Jos C. M. Baeten. “A brief history of process algebra”. In: *Theor. Comput. Sci.* 335.2-3 (2005), pp. 131–146.
- [21] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008, pp. I–XVII, 1–975. ISBN: 978-0-262-02649-9.
- [22] Matteo Baldoni et al. “Choice, interoperability, and conformance in interaction protocols and service choreographies”. In: *AAMAS (2)*. Ed. by Carles Sierra et al. IFAAMAS, 2009, pp. 843–850. ISBN: 978-0-9817381-7-8.
- [23] Matteo Baldoni et al. “Modeling agents in a logic action language”. In: *Proc. of Workshop on Practical Reasoning Agents, FAPR*. Citeseer. 2000.
- [24] Matteo Baldoni et al. “Verifying Protocol Conformance for Logic-Based Communicating Agents”. In: *CLIMA*. Ed. by João Alexandre Leite and Paolo Torroni. Vol. 3487. Lecture Notes in Computer Science. Springer, 2004, pp. 196–212. ISBN: 3-540-28060-X.
- [25] Matteo Baldoni et al. “Verifying the Conformance of Web Services to Global Interaction Protocols: A First Step”. In: *EPEW/WS-FM*. Ed. by Mario Bravetti, Leïla Kloul, and Gianluigi Zavattaro. Vol. 3670. Lecture Notes in Computer Science. Springer, 2005, pp. 257–271. ISBN: 3-540-28701-9.
- [26] Arindam Banerji et al. *Web Services Conversation Language (WSCL) 1.0*. W3C Note. Mar. 2002. URL: <http://www.w3.org/TR/wsc110/>.
- [27] Adam Barker, Christopher D. Walton, and David Robertson. “Choreographing Web Services”. In: *IEEE T. Services Computing 2.2* (2009), pp. 152–166.
- [28] Alistair Barros, Marlon Dumas, and Phillipa Oaks. *A Critical Overview of the Web Services Choreography Description Language (WS-CDL)*. Tech. rep. www.bptrends.com, 2005.
- [29] Alistair P. Barros, Marlon Dumas, and Arthur H. M. ter Hofstede. “Service Interaction Patterns”. In: *Business Process Management*. Ed. by Wil M. P. van der Aalst et al. Vol. 3649. 2005, pp. 302–318. ISBN: 3-540-28238-6.
- [30] Alistair P. Barros, Marlon Dumas, and Phillipa Oaks. “Standards for Web Service Choreography and Orchestration: Status and Perspectives”. In: *Business Process Management Workshops*. Vol. 3812. 2005, pp. 61–74. ISBN: 3-540-32595-6.
- [31] Christian Bartelt. “An Optimistic Three-way Merge Based on a Meta-Model Independent Modularization of Models to Support Concurrent Evolution”. In: *MODSE 08: Proceedings of the 2nd Workshop on Model-Driven Software Evolution*. Athens, Greece: IEEE, 2008.

- [32] Christian Bartelt. “Consistence Preserving Model Merge in Collaborative Development Processes”. In: *Proceedings of the 2008 International Workshop on Comparison and Versioning of Software Models*. CVSM '08. Leipzig, Germany: ACM, 2008, pp. 13–18. ISBN: 978-1-60558-045-6. DOI: 10.1145/1370152.1370157. URL: <http://doi.acm.org/10.1145/1370152.1370157>.
- [33] Massimo Bartoletti, Emilio Tuosto, and Roberto Zunino. “On the Realizability of Contracts in Dishonest Systems”. In: *COORDINATION*. Ed. by Marjan Sirjani. Vol. 7274. Lecture Notes in Computer Science. Springer, 2012, pp. 245–260. ISBN: 978-3-642-30828-4.
- [34] Samik Basu and Tevfik Bultan. “Choreography conformance via synchronizability”. In: *WWW*. Ed. by Sadagopan Srinivasan et al. ACM, 2011, pp. 795–804. ISBN: 978-1-4503-0632-4.
- [35] Samik Basu, Tevfik Bultan, and Meriem Ouederni. “Deciding choreography realizability”. In: *POPL*. Ed. by John Field and Michael Hicks. ACM, 2012, pp. 191–202. ISBN: 978-1-4503-1083-3.
- [36] Samik Basu, Tevfik Bultan, and Meriem Ouederni. “Synchronizability for Verification of Asynchronously Communicating Systems”. In: *VMCAI*. Ed. by Viktor Kuncak and Andrey Rybalchenko. Vol. 7148. Lecture Notes in Computer Science. Springer, 2012, pp. 56–71. ISBN: 978-3-642-27939-3.
- [37] Hanène Ben-Abdallah and Stefan Leue. “Syntactic Detection of Process Divergence and Non-local Choice in Message Sequence Charts”. In: *TACAS*. Ed. by Ed Brinksma. Vol. 1217. Lecture Notes in Computer Science. Springer, 1997, pp. 259–274. ISBN: 3-540-62790-1.
- [38] Boualem Benatallah et al. “On Temporal Abstractions of Web Service Protocols”. In: *CAiSE Short Paper Proceedings*. Ed. by Orlando Belo et al. Vol. 161. CEUR Workshop Proceedings. CEUR-WS.org, 2005.
- [39] Ian Betteridge. *TechCrunch: Irresponsible journalism*. <http://www.technovia.co.uk/2009/02/techcrunch-irresponsible-journalism.html>. Blog. 2009.
- [40] Kamal Bhattacharya et al. “Towards Formal Analysis of Artifact-Centric Business Process Models”. In: *BPM*. Ed. by Gustavo Alonso, Peter Dadam, and Michael Rosemann. Vol. 4714. Lecture Notes in Computer Science. Springer, 2007, pp. 288–304. ISBN: 978-3-540-75182-3.
- [41] Laura Bocchi, Julien Lange, and Emilio Tuosto. “Amending Contracts for Choreographies”. In: *ICE*. Ed. by Alexandra Silva et al. Vol. 59. EPTCS. 2011, pp. 111–129.
- [42] Laura Bocchi et al. “A Theory of Design-by-Contract for Distributed Multiparty Interactions”. In: *CONCUR*. Ed. by Paul Gastin and François Laroussinie. Vol. 6269. Lecture Notes in Computer Science. Springer, 2010, pp. 162–176. ISBN: 978-3-642-15374-7.
- [43] Chiara Bodei and Gian Luigi Ferrari. “Choreography Rehearsal”. In: *WS-FM*. Ed. by Cosimo Laneve and Jianwen Su. Vol. 6194. Lecture Notes in Computer Science. Springer, 2009, pp. 29–45. ISBN: 978-3-642-14457-8.
- [44] Benedikt Bollig and Loïc Hélouët. “Realizability of Dynamic MSC Languages”. In: *CSR*. Ed. by Farid M. Ablayev and Ernst W. Mayr. Vol. 6072. Lecture Notes in Computer Science. Springer, 2010, pp. 48–59. ISBN: 978-3-642-13181-3.
- [45] Paul Bouché. “WS-CDL and Pi-Calculus”. In: *Business Process Management II—Winter Term 2006 (2005)*, pp. 56–71.
- [46] Mario Bravetti, Leïla Kloul, and Gianluigi Zavattaro, eds. *Formal Techniques for Computer Systems and Business Processes, European Performance Engineering Workshop, EPEW 2005 and International Workshop on Web Services and Formal Methods, WS-FM 2005, Versailles, France, September 1-3, 2005, Proceedings*. Vol. 3670. Lecture Notes in Computer Science. Springer, 2005. ISBN: 3-540-28701-9.

- [47] Mario Bravetti, Manuel Núñez, and Gianluigi Zavattaro, eds. *Web Services and Formal Methods, Third International Workshop, WS-FM 2006 Vienna, Austria, September 8-9, 2006, Proceedings*. Vol. 4184. Lecture Notes in Computer Science. Springer, 2006. ISBN: 3-540-38862-1.
- [48] Mario Bravetti and Gianluigi Zavattaro. “Contract Based Multi-party Service Composition”. In: *FSEN*. Ed. by Farhad Arbab and Marjan Sirjani. Vol. 4767. Lecture Notes in Computer Science. Springer, 2007, pp. 207–222. ISBN: 978-3-540-75697-2.
- [49] Mario Bravetti and Gianluigi Zavattaro. “Contract Compliance and Choreography Conformance in the Presence of Message Queues”. In: *WS-FM*. Ed. by Roberto Bruni and Karsten Wolf. Vol. 5387. Lecture Notes in Computer Science. Springer, 2008, pp. 37–54. ISBN: 978-3-642-01363-8.
- [50] Franck van Breugel and Maria Koshika. *Models and Verification of BPEL*. <http://www.cse.yorku.ca/~franck/research/drafts/tutorial.pdf>. 2005.
- [51] Tevfik Bultan and Xiang Fu. “Choreography Modeling and Analysis with Collaboration Diagrams”. In: *IEEE Data Eng. Bull.* 31.3 (2008), pp. 27–30.
- [52] Tevfik Bultan and Xiang Fu. “Specification of realizable service conversations using collaboration diagrams”. In: *Service Oriented Computing and Applications* 2.1 (2008), pp. 27–39.
- [53] Tevfik Bultan, Xiang Fu, and Jianwen Su. “Analyzing Conversations: Realizability, Synchronizability, and Verification”. In: *Test and Analysis of Web Services*. Ed. by Luciano Baresi and Elisabetta Di Nitto. Springer, 2007, pp. 57–85. ISBN: 978-3-540-72912-9.
- [54] ebXML Business Process Project Team. *ebXML Business Process Specification Schema Version 1.01*. Technical Specification. OASIS, May 2001.
- [55] Chao Cai et al. “A Formal Model for Channel Passing in Web Service Composition”. In: *IEEE SCC (2)*. IEEE Computer Society, 2008, pp. 495–496. ISBN: 978-0-7695-3283-7.
- [56] Luís Caires and Hugo Torres Vieira. “Conversation types”. In: *Theor. Comput. Sci.* 411.51-52 (2010), pp. 4399–4440.
- [57] David Callahan. “The Program Summary Graph and Flow-Sensitive Interprocedural Data Flow Analysis”. In: *PLDI*. 1988, pp. 47–56.
- [58] Sara Capecchi, Elena Giachino, and Nobuko Yoshida. “Global Escape in Multiparty Sessions”. In: *FSTTCS*. Ed. by Kamal Lodaya and Meena Mahajan. Vol. 8. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2010, pp. 338–351. ISBN: 978-3-939897-23-1.
- [59] Marco Carbone. “Session-based Choreography with Exceptions”. In: *Electr. Notes Theor. Comput. Sci.* 241 (2009), pp. 35–55.
- [60] Marco Carbone, Kohei Honda, and Nobuko Yoshida. “A Calculus of Global Interaction based on Session Types”. In: *Electr. Notes Theor. Comput. Sci.* 171.3 (2007), pp. 127–151.
- [61] Marco Carbone and Fabrizio Montesi. “Deadlock-freedom-by-design: multiparty asynchronous global programming”. In: *POPL*. Ed. by Roberto Giacobazzi and Radhia Cousot. ACM, 2013, pp. 263–274. ISBN: 978-1-4503-1832-7.
- [62] Marco Carbone et al. “A Logic for Choreographies”. In: *PLACES*. Ed. by Kohei Honda and Alan Mycroft. Vol. 69. EPTCS. 2010, pp. 29–43.
- [63] Jorge Cardoso et al. “Quality of service for workflows and web service processes”. In: *J. Web Sem.* 1.3 (2004), pp. 281–308.
- [64] Giuseppe Castagna, Nils Gesbert, and Luca Padovani. “A theory of contracts for Web services”. In: *ACM Trans. Program. Lang. Syst.* 31.5 (2009).
- [65] Yang Chang and Liu Fang. “A calculus for WS-CDL language”. In: *System Science, Engineering Design and Manufacturing Informatization (ICSEM), 2011 International Conference on*. Vol. 2. 2011, pp. 356–359. DOI: 10.1109/ICSSEM.2011.6081319.

- [66] Cai Chao and Qiu Zongyan. “An Approach to Check Choreography with Channel Passing in WS-CDL”. In: *Proceedings of the 2008 IEEE International Conference on Web Services*. ICWS '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 700–707. ISBN: 978-0-7695-3310-0. DOI: 10.1109/ICWS.2008.46. URL: <http://dx.doi.org/10.1109/ICWS.2008.46>.
- [67] Ned Chapin et al. “Types of software evolution and software maintenance”. In: *Journal of Software Maintenance* 13.1 (2001), pp. 3–30.
- [68] David A. Chappell. *Enterprise service bus*. O'Reilly Series. O'Reilly, 2004. ISBN: 9780596006754. URL: <http://books.google.de/books?id=Uhue3faV2mwC>.
- [69] Minas Charalambides, Peter Dinges, and Gul Agha. “Parameterized Concurrent Multi-Party Session Types”. In: *FOCLASA*. Ed. by Natallia Kokash and António Ravara. Vol. 91. EPTCS. 2012, pp. 16–30.
- [70] Vincenzo Ciancia et al. “Event based choreography”. In: *Sci. Comput. Program.* 75.10 (2010), pp. 848–878.
- [71] Vincenzo Ciancia et al. “Model-Driven Development of Long Running Transactions”. In: *Results of the SENSORIA Project*. Ed. by Martin Wirsing and Matthias M. Hölzl. Vol. 6582. Lecture Notes in Computer Science. Springer, 2011, pp. 326–348. ISBN: 978-3-642-20400-5.
- [72] David Cohn and Richard Hull. “Business Artifacts: A Data-centric Approach to Modeling Business Operations and Processes”. In: *IEEE Data Eng. Bull.* 32.3 (2009), pp. 3–9.
- [73] “Collins English Dictionary - Complete & Unabridged”. In: (2003). URL: <http://dictionary.reference.com/browse/x>.
- [74] Keith D. Cooper, Timothy J. Harvey, and Ken Kenned. “A Simple, Fast Dominance Algorithm”. In: *Software – Practice and Experience* 4.1 (2001), pp. 1–10.
- [75] Mario Cortes Cornax, Sophie Dupuy-Chessa, and Dominique Rieu. “Choreographies in BPMN 2.0: New Challenges and Open Questions”. In: *ZEUS*. Ed. by Andreas Schönberger, Oliver Kopp, and Niels Lohmann. Vol. 847. CEUR Workshop Proceedings. CEUR-WS.org, 2012, pp. 50–57.
- [76] Mario Cortes Cornax et al. “Evaluating Choreographies in BPMN 2.0 Using an Extended Quality Framework”. In: *BPMN*. Ed. by Remco M. Dijkman, Jörg Hofstetter, and Jana Koehler. Vol. 95. Lecture Notes in Business Information Processing. Springer, 2011, pp. 103–117. ISBN: 978-3-642-25159-7.
- [77] Steven E. Czerwinski et al. “An Architecture for a Secure Service Discovery Service”. In: *MOBICOM*. 1999, pp. 24–35.
- [78] Florian Daniel and Barbara Pernici. “Insights into Web Service Orchestration and Choreography”. In: *IJEER* 2.1 (2006), pp. 58–77.
- [79] Gero Decker. “Design and Analysis of Process Choreographies”. PhD thesis. Hasso Plattner Institute, 2009.
- [80] Gero Decker, Oliver Kopp, and Alistair P. Barros. “An Introduction to Service Choreographies (Servicechoreographien - eine Einführung)”. In: *it - Information Technology* 50.2 (2008), pp. 122–127.
- [81] Gero Decker and Mathias Weske. “Local Enforceability in Interaction Petri nets”. In: *BPM*. Ed. by Gustavo Alonso, Peter Dadam, and Michael Rosemann. Vol. 4714. Lecture Notes in Computer Science. Springer, 2007, pp. 305–319. ISBN: 978-3-540-75182-3.
- [82] Gero Decker, Johannes Maria Zaha, and Marlon Dumas. “Execution Semantics for Service Choreographies”. In: *WS-FM*. Ed. by Mario Bravetti, Manuel Núñez, and Gianluigi Zavattaro. Vol. 4184. Lecture Notes in Computer Science. Springer, 2006, pp. 163–177. ISBN: 3-540-38862-1.

- [83] Gero Decker et al. “BPEL4Chor: Extending BPEL for Modeling Choreographies”. In: *ICWS*. IEEE Computer Society, 2007, pp. 296–303.
- [84] Gero Decker et al. *Maestro for Let’s Dance: An environment for modeling service interactions*. Tech. rep. Queensland University of Technology, 2006.
- [85] Gero Decker et al. “Modeling Service Choreographies Using BPMN and BPEL4Chor”. In: *CAiSE*. Ed. by Zohra Bellahsene and Michel Léonard. Vol. 5074. Lecture Notes in Computer Science. Springer, 2008, pp. 79–93. ISBN: 978-3-540-69533-2.
- [86] Gero Decker et al. “Non-desynchronizable Service Choreographies”. In: *ICSOC*. Vol. 5364. Lecture Notes in Computer Science. 2008, pp. 331–346. ISBN: 978-3-540-89647-0.
- [87] Pierre-Malo Deniérou and Nobuko Yoshida. “Multiparty Session Types Meet Communicating Automata”. In: *ESOP*. Ed. by Helmut Seidl. Vol. 7211. Lecture Notes in Computer Science. Springer, 2012, pp. 194–213. ISBN: 978-3-642-28868-5.
- [88] Nirmal Desai and Munindar P. Singh. “On the Enactability of Business Protocols”. In: *AAAI*. Ed. by Dieter Fox and Carla P. Gomes. AAAI Press, 2008, pp. 1126–1131. ISBN: 978-1-57735-368-3.
- [89] Nirmal Desai et al. “Interaction Protocols as Design Abstractions for Business Processes”. In: *IEEE Trans. Software Eng.* 31.12 (2005), pp. 1015–1027.
- [90] Mariangiola Dezani-Ciancaglini and Ugo de’Liguoro. “Sessions and Session Types: An Overview”. In: *WS-FM*. Ed. by Cosimo Laneve and Jianwen Su. Vol. 6194. Lecture Notes in Computer Science. Springer, 2009, pp. 1–28. ISBN: 978-3-642-14457-8.
- [91] Gregorio Díaz et al. “Automatic Translation of WS-CDL Choreographies to Timed Automata”. In: *EPEW/WS-FM*. Ed. by Mario Bravetti, Leila Kloul, and Gianluigi Zavattaro. Vol. 3670. Lecture Notes in Computer Science. Springer, 2005, pp. 230–242. ISBN: 3-540-28701-9.
- [92] Remco M. Dijkman and Pieter Van Gorp. “BPMN 2.0 Execution Semantics Formalized as Graph Rewrite Rules”. In: *BPMN*. Ed. by Jan Mendling, Matthias Weidlich, and Mathias Weske. Vol. 67. Lecture Notes in Business Information Processing. Springer, 2010, pp. 16–30. ISBN: 978-3-642-16297-8.
- [93] Hywel R. Dunn-Davies and Jim Cunningham. “Verifying realizability and reachability in recursive interaction protocol specifications”. In: *AAMAS (2)*. Ed. by Carles Sierra et al. IFAAMAS, 2009, pp. 1233–1234. ISBN: 978-0-9817381-7-8.
- [94] Daniel Eichhorn, Agnes Koschmider, and Huayu Zhang, eds. *3rd Central-European Workshop on Services and their Composition, Services und ihre Komposition, ZEUS 2011, Karlsruhe, Germany, February 21-22, 2011. Proceedings*. Vol. 705. CEUR Workshop Proceedings. CEUR-WS.org, 2011.
- [95] Javier Esparza. “Decidability and Complexity of Petri Net Problems - An Introduction”. In: *Petri Nets*. Ed. by Wolfgang Reisig and Grzegorz Rozenberg. Vol. 1491. Lecture Notes in Computer Science. Springer, 1996, pp. 374–428. ISBN: 3-540-65306-6.
- [96] Dirk Fahland et al. “Many-to-Many: Some Observations on Interactions in Artifact Choreographies”. In: *ZEUS*. Ed. by Daniel Eichhorn, Agnes Koschmider, and Huayu Zhang. Vol. 705. CEUR Workshop Proceedings. CEUR-WS.org, 2011, pp. 9–15.
- [97] *Web Services Coordination (WS-Coordination) Version 1.2*. Tech. rep. <http://docs.oasis-open.org/ws-tx/wstx-wscoor-1.2-spec-os/wstx-wscoor-1.2-spec-os.html>. Feb. 2009.
- [98] Cédric Fournet et al. “Stuck-Free Conformance”. In: *CAV*. Ed. by Rajeev Alur and Doron Peled. Vol. 3114. Lecture Notes in Computer Science. Springer, 2004, pp. 242–254. ISBN: 3-540-22342-8.
- [99] Martin Fowler. *UML distilled: A Brief Guide to the Standard Object Modeling Language*. 3rd. Addison-Wesley Professional, 2003.

- [100] Xiang Fu. “Formal specification and verification of asynchronously communicating web services”. PhD thesis. University of California, Sept. 2004.
- [101] Xiang Fu, Tefvik Bultan, and Jianwen Su. “Analysis of interacting BPEL web services”. In: *WWW*. Ed. by Stuart I. Feldman et al. ACM, 2004, pp. 621–630. ISBN: 1-58113-844-X.
- [102] Xiang Fu, Tefvik Bultan, and Jianwen Su. “Realizability of Conversation Protocols with Message Contents”. In: *Int. J. Web Service Res.* 2.4 (2005), pp. 68–93.
- [103] Xiang Fu, Tefvik Bultan, and Jianwen Su. “Synchronizability of Conversations among Web Services”. In: *IEEE Trans. Software Eng.* 31.12 (2005), pp. 1042–1055.
- [104] Nicolas Genon, Patrick Heymans, and Daniel Amyot. “Analysing the Cognitive Effectiveness of the BPMN 2.0 Visual Notation”. In: *SLE*. Ed. by Brian A. Malloy, Steffen Staab, and Mark van den Brand. Vol. 6563. Lecture Notes in Computer Science. Springer, 2010, pp. 377–396. ISBN: 978-3-642-19439-9.
- [105] Rob J. van Glabbeek and W. P. Weijland. “Branching Time and Abstraction in Bisimulation Semantics”. In: *J. ACM* 43.3 (1996), pp. 555–600.
- [106] Alexander Grosskopf, Gero Decker, and Mathias Weske. *The Process: Business Process Modeling using BPMN*. 1st. Meghan-Kiffer Press, 2009, p. 182. ISBN: 978-0929652269.
- [107] Jens Gulden. *Explication of Termination Semantics as a Security-Relevant Feature in Business Process Modeling Languages*.
- [108] Christian Gutschier et al. “A Pitfall with BPMN Execution”. In: *WEB 2014, The Second International Conference on Building and Exploring Web Based Environments*. 2014, pp. 7–13.
- [109] Sylvain Hallé and Tefvik Bultan. “Realizability analysis for message-based interactions using shared-state projections”. In: *SIGSOFT FSE*. Ed. by Gruia-Catalin Roman and Kevin J. Sullivan. ACM, 2010, pp. 27–36. ISBN: 978-1-60558-791-2.
- [110] Alan R. Hevner et al. “Design Science in Information Systems Research”. In: *MIS Quarterly* 28.1 (2004), pp. 75–105. URL: <http://misq.org/design-science-in-information-systems-research.html>.
- [111] Marcel Hiel, Huib Aldewereld, and Frank Dignum. “Ensuring conformance in an evolving choreography”. In: *SOCA*. IEEE, 2010, pp. 1–4.
- [112] C. A. R. Hoare. “A Model for Communicating Sequential Processes”. In: *On the Construction of Programs*. 1980, pp. 229–254.
- [113] Birgit Hofreiter. “Registering UML models for global and local choreographies”. In: *ICEC*. Ed. by Dieter Fensel and Hannes Werthner. Vol. 342. ACM International Conference Proceeding Series. ACM, 2008, p. 37. ISBN: 978-1-60558-075-3.
- [114] Christian Huemer et al. *UN/CEFACTs Modeling Methodology (UMM), UMM Meta Model Foundation Module Version 1.0*. Technical Specification. UN/CEFACT, 2006.
- [115] Richard Hull. “Artifact-Centric Business Process Models: Brief Survey of Research Results and Challenges”. In: *OTM Conferences (2)*. Ed. by Robert Meersman and Zahir Tari. Vol. 5332. Lecture Notes in Computer Science. Springer, 2008, pp. 1152–1163. ISBN: 978-3-540-88872-7.
- [116] Raman Kazhamiakin and Marco Pistore. “Analysis of Realizability Conditions for Web Service Choreographies”. In: *FORTE*. Ed. by Elie Najm, Jean-François Pradat-Peyre, and Véronique Donzeau-Gouge. Vol. 4229. Lecture Notes in Computer Science. Springer, 2006, pp. 61–76. ISBN: 3-540-46219-8.
- [117] Raman Kazhamiakin and Marco Pistore. “Choreography Conformance Analysis: Asynchronous Communications and Information Alignment”. In: *WS-FM*. Ed. by Mario Bravetti, Manuel Núñez, and Gianluigi Zavattaro. Vol. 4184. Lecture Notes in Computer Science. Springer, 2006, pp. 227–241. ISBN: 3-540-38862-1.

- [118] Raman Kazhamiakin, Marco Pistore, and Luca Santuari. “Analysis of communication models in web service compositions”. In: *WWW*. Ed. by Les Carr et al. ACM, 2006, pp. 267–276. ISBN: 1-59593-323-9.
- [119] David Knuplesch, Rüdiger Pryss, and Manfred Reichert. “A Formal Framework for Data-Aware Process Interaction Models”. In: (2012).
- [120] Oliver Kopp, Tammo van Lessen, and Jörg Nitzsche. “The Need for a Choreography-aware Service Bus”. English. In: *YR-SOC 2008*. Online, June 2008, pp. 28–34. URL: http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2008-38&engl=0.
- [121] Oliver Kopp, Matthias Wieland, and Frank Leymann. “Towards choreography transactions”. In: *ZEUS*. Ed. by Oliver Kopp and Niels Lohmann. Vol. 438. CEUR Workshop Proceedings. CEUR-WS.org, 2009, pp. 49–54.
- [122] Oliver Kopp et al. “Interaction Choreography Models in BPEL: Choreographies on the Enterprise Service Bus”. In: *S-BPM ONE*. Ed. by Albert Fleischmann et al. Vol. 138. Communications in Computer and Information Science. Springer, 2010, pp. 36–53. ISBN: 978-3-642-23134-6.
- [123] Bogdan Korel and Janusz W. Laski. “Dynamic Program Slicing”. In: *Inf. Process. Lett.* 29.3 (1988), pp. 155–163. DOI: 10.1016/0020-0190(88)90054-3. URL: [http://dx.doi.org/10.1016/0020-0190\(88\)90054-3](http://dx.doi.org/10.1016/0020-0190(88)90054-3).
- [124] Ivan Lanese, Fabrizio Montesi, and Gianluigi Zavattaro. “Amending Choreographies”. In: *CoRR* abs/1308.0390, abs/1308.0390 (2013).
- [125] Cosimo Laneve and Jianwen Su, eds. *Web Services and Formal Methods, 6th International Workshop, WS-FM 2009, Bologna, Italy, September 4-5, 2009, Revised Selected Papers*. Vol. 6194. Lecture Notes in Computer Science. Springer, 2010. ISBN: 978-3-642-14457-8.
- [126] Hong Anh Le and Ninh Thuan Truong. “Modeling and Verifying WS-CDL Using Event-B”. In: *ICCASA*. Ed. by Phan Cong Vinh et al. Vol. 109. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering. Springer, 2012, pp. 290–299. ISBN: 978-3-642-36641-3.
- [127] Leonardo A. F. Leite et al. “A systematic literature review of service choreography adaptation”. In: *Service Oriented Computing and Applications* 7.3 (2013), pp. 199–216.
- [128] George Lekeas, Christos Kloukinas, and Kostas Stathis. “Producing Enactable Protocols in Artificial Agent Societies”. In: *PRIMA*. Ed. by David Kinny et al. Vol. 7047. Lecture Notes in Computer Science. Springer, 2011, pp. 311–322. ISBN: 978-3-642-25043-9.
- [129] Jing Li, Huibiao Zhu, and Geguang Pu. “Conformance Validation between Choreography and Orchestration”. In: *TASE*. IEEE Computer Society, 2007, pp. 473–482.
- [130] Jing Li et al. “Modeling and Verifying Web Services Choreography Using Process Algebra”. In: *SEW*. IEEE Computer Society, 2007, pp. 256–268.
- [131] Mark Little. *A Second WS-CDL Tool-Suite Is Born*. Nov. 2007. URL: <http://www.infoq.com/news/2007/11/wscomposition>.
- [132] Niels Lohmann. “Communication models for services”. In: *ZEUS*. Ed. by Christian Gierds and Jan Sürmeli. Vol. 563. CEUR Workshop Proceedings. CEUR-WS.org, 2010, pp. 9–16.
- [133] Niels Lohmann and Karsten Wolf. “Artifact-Centric Choreographies”. In: *ICSOC*. Ed. by Paul P. Maglio et al. Vol. 6470. Lecture Notes in Computer Science. 2010, pp. 32–46. ISBN: 978-3-642-17357-8.
- [134] Niels Lohmann and Karsten Wolf. “Decidability Results for Choreography Realization”. In: *ICSOC*. Ed. by Gerti Kappel, Zakaria Maamar, and Hamid R. Motahari-Nezhad. Vol. 7084. Lecture Notes in Computer Science. Springer, 2011, pp. 92–107. ISBN: 978-3-642-25534-2.

- [135] Niels Lohmann and Karsten Wolf. “Realizability Is Controllability”. In: *WS-FM*. Ed. by Cosimo Laneve and Jianwen Su. Vol. 6194. Lecture Notes in Computer Science. Springer, 2009, pp. 110–127. ISBN: 978-3-642-14457-8.
- [136] Hugo A. López, Carlos Olarte, and Jorge A. Pérez. “Towards a Unified Framework for Declarative Structured Communications”. In: *PLACES*. Ed. by Alastair R. Beresford and Simon J. Gay. Vol. 17. EPTCS. 2009, pp. 1–15.
- [137] Ayman Mahfouz et al. “Requirements-Driven Collaborative Choreography Customization”. In: *ICSOC/ServiceWave*. Ed. by Luciano Baresi, Chi-Hung Chi, and Jun Suzuki. Vol. 5900. Lecture Notes in Computer Science. 2009, pp. 144–158. ISBN: 978-3-642-10382-7.
- [138] Michele Mancioppi et al. “Sound Multi-party Business Protocols for Service Networks”. In: *ICSOC*. Vol. 5364. Lecture Notes in Computer Science. 2008, pp. 302–316. ISBN: 978-3-540-89647-0.
- [139] Michele Mancioppi et al. “Towards a Quality Model for Choreography”. In: *ICSOC/ServiceWave Workshops*. Ed. by Asit Dan, Frederic Gittler, and Farouk Toumani. Vol. 6275. Lecture Notes in Computer Science. 2009, pp. 435–444. ISBN: 978-3-642-16131-5.
- [140] David Martin et al. *OWL-S: Semantic Markup for Web Services*. W3C Member Submission. Nov. 2004. URL: <http://www.w3.org/Submission/OWL-S>.
- [141] Sjouke Mauw and Michel A. Reniers. “High-level message sequence charts”. In: *SDL Forum*. Ed. by Ana R. Cavalli and Amardeo Sarma. Elsevier, 1997, pp. 291–306.
- [142] Stephen McIlvenna, Marlon Dumas, and Moe Thandar Wynn. “Synthesis of Orchestrators from Service Choreographies”. In: *APCCM*. Ed. by Markus Kirchberg and Sebastian Link. Vol. 96. CRPIT. Australian Computer Society, 2009, pp. 129–138. ISBN: 978-1-920682-77-4.
- [143] Ashley T. McNeile. “Protocol contracts with application to choreographed multiparty collaborations”. In: *Service Oriented Computing and Applications 4.2* (2010), pp. 109–136.
- [144] Robert Meersman and Zahir Tari, eds. *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE, OTM Confederated International Conferences, CoopIS, DOA, GADA, and ODBASE 2006, Montpellier, France, October 29 - November 3, 2006. Proceedings, Part I*. Vol. 4275. Lecture Notes in Computer Science. Springer, 2006. ISBN: 3-540-48287-3.
- [145] Jan Mendling and Michael Hafner. “From Inter-organizational Workflows to Process Execution: Generating BPEL from WS-CDL”. In: *OTM Workshops*. Ed. by Robert Meersman et al. Vol. 3762. Lecture Notes in Computer Science. Springer, 2005, pp. 506–515. ISBN: 3-540-29739-1.
- [146] Jan Mendling and Michael Hafner. “From WS-CDL choreography to BPEL process orchestration”. In: *J. Enterprise Inf. Management* 21.5 (2008), pp. 525–542.
- [147] *Message Sequence Chart*. Recommendation Z.120. ITU-T, Feb. 2011.
- [148] Robert B. Miller. “Response time in man-computer conversational transactions”. In: *American Federation of Information Processing Societies: Proceedings of the AFIPS '68 Fall Joint Computer Conference, December 9-11, 1968, San Francisco, California, USA - Part I*. Vol. 33. AFIPS Conference Proceedings. AFIPS / ACM / Thomson Book Company, Washington D.C., 1968, pp. 267–277. DOI: 10.1145/1476589.1476628. URL: <http://doi.acm.org/10.1145/1476589.1476628>.
- [149] Robin Milner. *A Calculus of Communicating Systems*. Vol. 92. Lecture Notes in Computer Science. Springer, 1980. ISBN: 3-540-10235-3.
- [150] Robin Milner. *Communicating and mobile systems - the Pi-calculus*. Cambridge University Press, 1999, pp. I–XII, 1–161. ISBN: 978-0-521-65869-0.
- [151] Robin Milner. *Communication and concurrency*. PHI Series in computer science. Prentice Hall, 1989, pp. I–XI, 1–260. ISBN: 978-0-13-115007-2.

- [152] Marco Montali et al. “Declarative specification and verification of service choreographies”. In: *TWEB* 4.1 (2010).
- [153] Carlo Montangero and Laura Semini. “A Logical View of Choreography”. In: *COORDINATION*. Ed. by Paolo Ciancarini and Herbert Wiklicky. Vol. 4038. Lecture Notes in Computer Science. Springer, 2006, pp. 179–193. ISBN: 3-540-34694-5.
- [154] Carlo Montangero and Laura Semini. “Distributed States Temporal Logic”. In: *CoRR* cs.LO/0304046 (2003).
- [155] Arjan J. Mooij, Nicolae Goga, and Judi Romijn. “Non-local Choice and Beyond: Intricacies of MSC Choice Nodes”. In: *FASE*. Ed. by Maura Cerioli. Vol. 3442. Lecture Notes in Computer Science. Springer, 2005, pp. 273–288. ISBN: 3-540-25420-X.
- [156] Arjan J. Mooij, Judi Romijn, and Wieger Wesseling. “Realizability Criteria for Compositional MSC”. In: *AMAST*. Ed. by Michael Johnson and Varmo Vene. Vol. 4019. Lecture Notes in Computer Science. Springer, 2006, pp. 248–262. ISBN: 3-540-35633-9.
- [157] Abdolmajid Mousavi et al. “Strong Safe Realizability of Message Sequence Chart Specifications”. In: *FSEN*. Ed. by Farhad Arbab and Marjan Sirjani. Vol. 4767. Lecture Notes in Computer Science. Springer, 2007, pp. 334–349. ISBN: 978-3-540-75697-2.
- [158] Markus Müller-Olm. “Precise interprocedural dependence analysis of parallel programs”. In: *Theor. Comput. Sci.* 311.1-3 (2004), pp. 325–388.
- [159] Eugene W. Myers. “A Precise Interprocedural Data Flow Algorithm”. In: *POPL*. 1981, pp. 219–230.
- [160] Flemming Nielson and Hanne Riis Nielson. “Interprocedural Control Flow Analysis”. In: *Programming Languages and Systems, 8th European Symposium on Programming, ESOP’99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS’99, Amsterdam, The Netherlands, 22-28 March, 1999, Proceedings*. Ed. by S. Doaitse Swierstra. Vol. 1576. Lecture Notes in Computer Science. Springer, 1999, pp. 20–39. ISBN: 3-540-65699-5. DOI: 10.1007/3-540-49099-X_3. URL: http://dx.doi.org/10.1007/3-540-49099-X_3.
- [161] James Odell, H Van Dyke Parunak, and Bernhard Bauer. “Extending UML for agents”. In: *Ann Arbor* 1001 (), p. 48103.
- [162] Gustavo Ansaldi Oliva et al. “Choreography Dynamic Adaptation Prototype”. In: (2012).
- [163] OMG. *Business Process Model and Notation Version 2.0*. OMG Specification formal/2011-01-03. Object Management Group, Jan. 2011.
- [164] OMG. *Business Process Modeling Notation Version 1.2*. OMG Specification. Object Management Group, Feb. 2008.
- [165] Mike P. Papazoglou. “Web Services and Business Transactions”. In: *World Wide Web* 6.1 (2003), pp. 49–91.
- [166] Chris Peltz. “Web Services Orchestration and Choreography”. In: *IEEE Computer* 36.10 (2003), pp. 46–52.
- [167] Amir Pnueli and Roni Rosner. “On the Synthesis of an Asynchronous Reactive Module”. In: *ICALP*. Ed. by Giorgio Ausiello, Mariangiola Dezani-Ciancaglini, and Simona Ronchi Della Rocca. Vol. 372. Lecture Notes in Computer Science. Springer, 1989, pp. 652–671. ISBN: 3-540-51371-X.
- [168] Pascal Poizat and Gwen Salaün. “Checking the realizability of BPMN 2.0 choreographies”. In: *SAC*. Ed. by Sascha Ossowski and Paola Lecca. ACM, 2012, pp. 1927–1934. ISBN: 978-1-4503-0857-1.
- [169] Julien Ponge et al. “Fine-Grained Compatibility and Replaceability Analysis of Timed Web Service Protocols”. In: *ER*. Ed. by Christine Parent et al. Vol. 4801. Lecture Notes in Computer Science. Springer, 2007, pp. 599–614. ISBN: 978-3-540-75562-3.

- [170] Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. “Precise Interprocedural Dataflow Analysis via Graph Reachability”. In: *POPL*. 1995, pp. 49–61.
- [171] Stefanie Rinderle, Andreas Wombacher, and Manfred Reichert. “Evolution of Process Choreographies in DYCHOR”. In: *OTM Conferences (1)*. Ed. by Robert Meersman and Zahir Tari. Vol. 4275. Lecture Notes in Computer Science. Springer, 2006, pp. 273–290. ISBN: 3-540-48287-3.
- [172] Ismael Rodríguez et al. “A centralized and a decentralized method to automatically derive choreography-conforming web service systems”. In: *J. Log. Algebr. Program.* 81.2 (2012), pp. 127–159.
- [173] Dumitru Roman, Michael Kifer, and Dieter Fensel. “WSMO Choreography: From Abstract State Machines to Concurrent Transaction Logic”. In: *ESWC*. Ed. by Sean Bechhofer et al. Vol. 5021. Lecture Notes in Computer Science. Springer, 2008, pp. 659–673. ISBN: 978-3-540-68233-2.
- [174] Dumitru Roman et al. *Ontology-based Choreography*. Final Draft D14v1.0. WSMO, Feb. 2007.
- [175] Nima Roohi, Gwen Salaün, and Seyed-Hassan Mirian-Hosseiniabadi. “Analyzing Chor Specifications by Translation into FSP”. In: *Electr. Notes Theor. Comput. Sci.* 255 (2009), pp. 159–176.
- [176] Steve Ross-Talbot and Tony Fletcher. *Web Services Choreography Description Language: Primer*. Working Draft. W3C, June 2006. URL: <http://www.w3.org/TR/ws-cdl-10-primer/>.
- [177] Valentín Valero Ruiz et al. “Transforming Web Services Choreographies with priorities and time constraints into prioritized-time colored Petri nets”. In: *Sci. Comput. Program.* 77.3 (2012), pp. 290–313.
- [178] Seung Hwan Ryu et al. “A Framework for Managing the Evolution of Business Protocols in Web Services”. In: *APCCM*. Vol. 67. CRPIT. Australian Computer Society, 2007, pp. 49–59. ISBN: 1-920-68248-1.
- [179] Gwen Salaün, Tevfik Bultan, and Nima Roohi. “Realizability of Choreographies Using Process Algebra Encodings”. In: *IEEE T. Services Computing* 5.3 (2012), pp. 290–304.
- [180] Gwen Salaün and Nima Roohi. “On Realizability and Dynamic Reconfiguration of Choreographies”. In: *Actas de los Talleres de las Jornadas de Ingeniería del Software y Bases de Datos 3.4* (2009), pp. 21–31.
- [181] Davide Sangiorgi and David Walker. *The Pi-Calculus - a theory of mobile processes*. Cambridge University Press, 2001, pp. I–XII, 1–580. ISBN: 978-0-521-78177-0.
- [182] Karsten Schmidt. “Controllability of Open Workflow Nets”. In: *EMISA*. Ed. by Jörg Desel and Ulrich Frank. Vol. 75. LNI. GI, 2005, pp. 236–249. ISBN: 3-88579-404-7.
- [183] Marc-Thomas Schmidt et al. “The Enterprise Service Bus: Making service-oriented architecture real”. In: *IBM Systems Journal* 44.4 (2005), pp. 781–798.
- [184] Andreas Schönberger. “Do We Need a Refined Choreography Notion?” In: *ZEUS*. Ed. by Daniel Eichhorn, Agnes Koschmider, and Huayu Zhang. Vol. 705. CEUR Workshop Proceedings. CEUR-WS.org, 2011, pp. 16–23.
- [185] David Schumm et al. “Essential Aspects of Compliance Management with Focus on Business Process Automation”. In: *ISSS/BPSC*. Ed. by Witold Abramowicz et al. Vol. 177. LNI. GI, 2010, pp. 127–138. ISBN: 978-3-88579-271-0.
- [186] *Service-Oriented Computing - ICSOC 2008, 6th International Conference, Sydney, Australia, December 1-5, 2008. Proceedings*. Vol. 5364. Lecture Notes in Computer Science. 2008. ISBN: 978-3-540-89647-0.

- [187] Carles Sierra et al., eds. *8th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2009), Budapest, Hungary, May 10-15, 2009, Volume 2*. IFAA-MAS, 2009. ISBN: 978-0-9817381-7-8.
- [188] Munindar P. Singh. “LoST: Local State Transfer - An Architectural Style for the Distributed Enactment of Business Protocols”. In: *ICWS*. IEEE Computer Society, 2011, pp. 57–64. ISBN: 978-1-4577-0842-8.
- [189] Saurabh Sinha and Mary Jean Harrold. “Analysis and Testing of Programs with Exception Handling Constructs”. In: *IEEE Trans. Software Eng.* 26.9 (2000), pp. 849–871.
- [190] Christian Stahl, Peter Massuthe, and Jan Bretschneider. “Deciding Substitutability of Services with Operating Guidelines”. In: *T. Petri Nets and Other Models of Concurrency*. Lecture Notes in Computer Science 2 (2009). Ed. by Kurt Jensen and Wil M. P. van der Aalst, pp. 172–191.
- [191] Jianwen Su et al. “Towards a Theory of Web Service Choreographies”. In: *WS-FM*. Ed. by Marlon Dumas and Reiko Heckel. Vol. 4937. Lecture Notes in Computer Science. Springer, 2007, pp. 1–16. ISBN: 978-3-540-79229-1.
- [192] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. “An Interaction-based Language and its Typing System”. In: *PARLE*. Ed. by Constantine Halatsis et al. Vol. 817. Lecture Notes in Computer Science. Springer, 1994, pp. 398–413. ISBN: 3-540-58184-7.
- [193] Robert Endre Tarjan. “Testing Flow Graph Reducibility”. In: *J. Comput. Syst. Sci.* 9.3 (1974), pp. 355–365.
- [194] Valentin Valero et al. “A Petri net approach for the design and analysis of Web Services Choreographies”. In: *J. Log. Algebr. Program.* 78.5 (2009), pp. 359–380.
- [195] W3C. *Web Services Choreography Description Language Version 1.0*. Candidate Recommendation. W3C, Nov. 2005.
- [196] *W3C Process Document*. Accessed on October 30, 2013. URL: <http://www.w3.org/Consortium/Process/>.
- [197] Hongbing Wang et al. “A model checker for WS-CDL”. In: *Journal of Systems and Software* 83.10 (2010), pp. 1651–1661.
- [198] Barbara Weber, Manfred Reichert, and Stefanie Rinderle-Ma. “Change patterns and change support features - Enhancing flexibility in process-aware information systems”. In: *Data Knowl. Eng.* 66.3 (2008), pp. 438–466.
- [199] Barbara Weber, Stefanie Rinderle, and Manfred Reichert. “Change Patterns and Change Support Features in Process-Aware Information Systems”. In: *CAiSE*. Vol. 4495. Lecture Notes in Computer Science. Springer, 2007, pp. 574–588. ISBN: 978-3-540-72987-7.
- [200] Matthias Weidlich, Mathias Weske, and Jan Mendling. “Change Propagation in Process Models Using Behavioural Profiles”. In: *IEEE SCC*. IEEE Computer Society, 2009, pp. 33–40.
- [201] Sebastian Wieczorek et al. “Precise Steps for Choreography Modeling for SOA Validation and Verification”. In: *SOSE*. Ed. by Jonathan Lee, Deron Liang, and Y. C. Cheng. IEEE Computer Society, 2008, pp. 148–153.
- [202] Andreas Wombacher. “Alignment of Choreography Changes in BPEL Processes”. In: *IEEE SCC*. IEEE Computer Society, 2009, pp. 1–8.
- [203] William A. Wulf. “A case against the GOTO”. In: Upper Saddle River, NJ, USA: Yourdon Press, 1979, pp. 83–98. ISBN: 0-917072-14-6. URL: <http://dl.acm.org/citation.cfm?id=1241515.1241523>.
- [204] Hongli Yang et al. “A Formal Model for Web Service Choreography Description Language (WS-CDL)”. In: *Proc. of ICWS 2006*. IEEE Computer Society, 2006.

- [205] Hongli Yang et al. “Exploring the Connection of Choreography and Orchestration with Exception Handling and Finalization/Compensation”. In: *FORTE*. Ed. by John Derrick and Jüri Vain. Vol. 4574. Lecture Notes in Computer Science. Springer, 2007, pp. 81–96. ISBN: 978-3-540-73195-5.
- [206] Hongli Yang et al. “Reasoning about Channel Passing in Choreography”. In: *TASE*. IEEE Computer Society, 2008, pp. 135–142. ISBN: 978-0-7695-3249-3.
- [207] Hongli Yang et al. “Type Checking Choreography Description Language”. In: *ICFEM*. Ed. by Zhiming Liu and Jifeng He. Vol. 4260. Lecture Notes in Computer Science. Springer, 2006, pp. 264–283. ISBN: 3-540-47460-9.
- [208] Wing Lok Yeung. “Mapping WS-CDL and BPEL into CSP for Behavioural Specification and Verification of Web Services”. In: *ECOWS*. IEEE Computer Society, 2006, pp. 297–305. ISBN: 0-7695-2737-X.
- [209] Johannes Maria Zaha et al. “Bridging Global and Local Models of Service-Oriented Systems”. In: *IEEE Transactions on Systems, Man, and Cybernetics, Part C* 38.3 (2008), pp. 302–318.
- [210] Johannes Maria Zaha et al. “Let’s Dance: A Language for Service Behavior Modeling”. In: *OTM Conferences (1)*. Ed. by Robert Meersman and Zahir Tari. Vol. 4275. Lecture Notes in Computer Science. Springer, 2006, pp. 145–162. ISBN: 3-540-48287-3.
- [211] Johannes Maria Zaha et al. “Service Interaction Modeling: Bridging Global and Local Views”. In: *EDOC*. IEEE Computer Society, 2006, pp. 45–55. ISBN: 0-7695-2558-X.
- [212] Xiangpeng Zhao, Hongli Yang, and Zongyan Qiu. “Towards the Formal Model and Verification of Web Service Choreography Description Language”. In: *WS-FM*. Ed. by Mario Bravetti, Manuel Núñez, and Gianluigi Zavattaro. Vol. 4184. Lecture Notes in Computer Science. Springer, 2006, pp. 273–287. ISBN: 3-540-38862-1.
- [213] Lei Zhou et al. “Automatically Testing Web Services Choreography with Assertions”. In: *ICFEM*. Ed. by Jin Song Dong and Huibiao Zhu. Vol. 6447. Lecture Notes in Computer Science. Springer, 2010, pp. 138–154. ISBN: 978-3-642-16900-7.

Terms and Abbreviations

- AAA** Awareness Annotation Algorithm. 90, 95, 96, 201
- ADG** Awareness Dependency Graph. 201
- API** Application Programming Interface. 201
- AUML** Agent UML. 23, 201
- AWM** Awareness Model. 83, 89, 90, 93–97, 99, 133, 153, 166–168, 171, 175, 201

- B2Bi** Business-to-Business Integration. 13, 201
- BAM** Business Activity Monitoring. 201
- BDV** Business Domain View. 201
- BNF** Backus-Naur Form. 56, 95, 165, 201
- BPM** Business Process Management. 1, 13, 14, 20, 25, 54, 85, 174, 182, 201
- BPMN v1.x** Business Process Modeling Notation. 30, 201
- BPMN v2.0** Business Process Model and Notation. 2, 3, 7, 18–20, 24, 25, 28, 30–34, 38, 41, 47, 55, 105, 112, 113, 174, 177, 201
- BPSS** Business Process Specification Schema. 25, 201
- BRV** Business Requirements View. 201
- BTV** Business Transaction View. 201
- BWA** Bird-Watching Algorithm. 201

- CCS** Calculus of Communicating Systems. 21, 201
- CDL** Choreography Description Language. 201
- CFG** Control Flow Graph. 28, 74–76, 80–83, 89, 102, 103, 201

- DOM** Document Object Model. 201
- DSL** Domain-Specific Language. 12, 20, 53, 56, 57, 76, 165, 201

- ebXML** Electronic Business using eXtensible Markup Language. 25, 201
- ESB** Enterprise Service Bus. 16, 56, 201

- FIFO** First in, first out. 201

- FSA** Finite State Automata. 46, 201
- FSM** Finite State Machine. 21–23, 41, 42, 45, 48, 49, 185, 201
- FSP** Finite State Process. 201
- HMSC** High-level Message Sequence Charts. 26, 27, 46, 201
- HTTP** Hypertext Transfer Protocol. 8, 201
- IDE** Integrated Development Environment. 6, 7, 12, 112, 165, 175, 201
- IETF** Internet Engineering Task Force. 201
- ITU** International Telecommunication Union. 25, 201
- JMS** Java Message Service. 8, 201
- JVM** Java Virtual Machine. 171, 201
- LTL** Linear Temporal Logics. 86, 201
- MAS** Multi-Agent Systems. 23, 201
- MDA** Model-Driven Architecture. 56, 201
- MDS** Mereotopology of Discrete Space. 201
- MSC** Message Sequence Chart. 25–27, 34, 35, 46, 201
- MSG** Message Sequence Graph. 201
- OASIS** Organization for the Advancement of Structured Information Standards. 24, 174, 201
- OMG** Object Management Group. 24, 34, 201
- OO** Object-Oriented. 201
- PAIS** Process-Aware Information Systems. 201
- PDG** Program Dependence Graph. 201
- PST** refined Process Structure Tree. 201
- QoS** Quality of Service. 4, 8, 16, 17, 46, 201
- RFC** Request For Comments. 201
- SBA** Service-Based Application. 201
- SD** State Diagram. 62, 201
- SESE** Single-Entry Single-Exit. 201
- SLA** Service-Level Agreement. 201
- SOA** Service-Oriented Architecture. 1, 13, 14, 16, 25, 28, 29, 54, 55, 85, 174, 201
- SOAP** Simple Object Access Protocol. 8, 201

- SOC** Service-Oriented Computing. 201
- STS** State-Transition System. 21, 23, 41, 45, 201
- UDDI** Universal Description Discovery and Integration. 17, 201
- UML 1.x** Unified Modeling Language version 1.x. 26, 201
- UML 2.x** Unified Modeling Language version 2.x. 23, 25–28, 34, 42, 44, 49, 51, 201
- UMM** UN/CEFACT Modeling Methodology. 201
- UN/CEFACT** United Nations Centre for Trade Facilitation and Electronic Business. 25, 201
- UPS** Uninterruptible Power Supply. 201
- URI** Uniform Resource Identifier. 56, 201
- URL** Uniform Resource Locator. 201
- UUID** Universally Unique Identifier. 201
- W3C** World Wide Web Consortium. 24, 27, 28, 174, 201
- WS** Web Service. 201
- WS-BPEL** Business Process Execution Language for Web Services. 7, 17, 18, 29, 53, 54, 58, 60, 201
- WS-CDL** Web Services Choreography Description Language. 18, 20, 23, 25, 27–29, 105, 174, 201
- WSCl** Web Service Choreography Interface. 25, 27, 201
- WSCL** Web Service Conversation Language. 25, 27, 201
- WSDL** Web Services Description Language. 21, 201
- WSMO** Web Service Modeling Ontology. 18, 25, 201
- XML** eXtensible Markup Language. 29, 30, 34, 201
- XSD** XML Schema Definition. 7, 201
- YAWL** Yet Another Workflow Language. 201