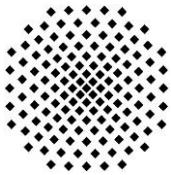


Universität Stuttgart

Fakultät Informatik, Elektrotechnik und Informationstechnik

Awareness-based Realizability Analysis of Service Choreographies

Michele Mancioppi, Olha Danylevych

Report 2012/01

Institut für Architektur von Anwendungssystemen

Universitätsstraße 38
70569 Stuttgart
Germany

CR: H.4.1

Awareness-based Realizability Analysis of Service Choreographies*

Michele Mancioppi & Olha Danylevych
Institute of Architecture of Application Systems (IAAS)
University of Stuttgart, Stuttgart, Germany
[firstname.lastname]@iaas.uni-stuttgart.de

1 Introduction

Service choreographies are *technical contracts* which specify the message-based interactions among collaborating parties, called *participants*. Figure 1 introduces the basic terminology related to service choreographies. The ordering and timing of the message-based interactions that a participant is tasked with performing in a choreography is called *role*. The participants implement in *participant implementations* the messaging behaviors required by the roles they are assigned. An *enactment* is the cumulative execution of the participants implementations and the resulting message exchanges. When their participant implementations collectively perform an enactment, the participants are said to *enact* the choreography.

There are multiple paradigms for specifying service choreographies, the most prominent of which are *interconnection* and *interaction*. In interconnection choreographies, each role is modeled explicitly as a *participant skeleton* [DKLW07, DKB08, DKLW09], i.e. an orchestration – possibly non-executable – that models the part of a possible internal behavior of the participant limited to the generation and consumption of messages. The participant skeletons are "wired" together by means of the message exchanges. Since each role is specified separately, interconnection choreographies may suffer from deadlocks that lead some of their enactments to get "stuck," i.e. the enactments reach states in which the participants cannot perform the actions required for the enactments to progress [Loh08, DW11].

The interaction paradigm for modeling choreographies foresees the specification of the messaging occurring among the participants from a global perspective, e.g. as a graph-based process in which the message exchanges among the participants are modeled as activities. Languages that model

*The research leading to these results has received funding from the European Community's Seventh Framework Programme under the Network of Excellence S-Cube - Grant Agreement n° 215483.

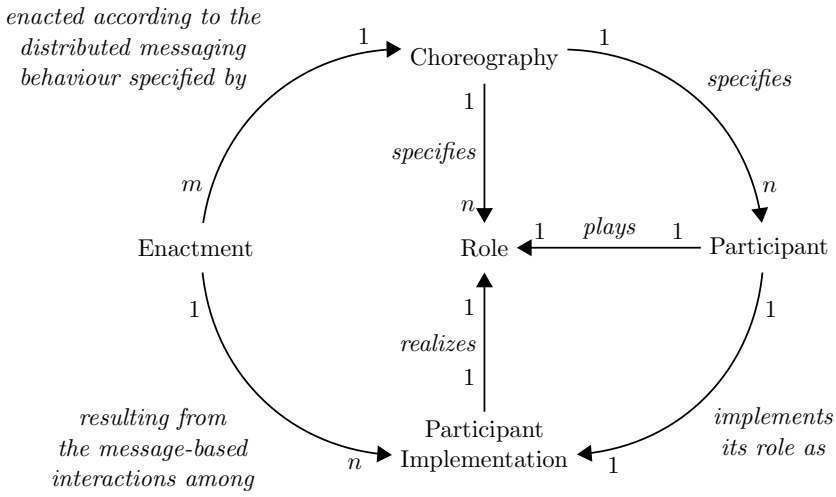


Figure 1: Terminology map of choreography-related terms.

choreographies using the interaction paradigm are, among others, the Business Process Model and Notation (BPMN v2.0) [OMG11] (choreography diagrams) and the Web Services Choreography Description Language (WS-CDL) [W3C05]. Since the ordering and timing of the message exchanges are specified from a global perspective, it is surprisingly simple to specify choreographies that cannot be correctly enacted by their participants. Consider for example a choreography that specifies that the participant p_1 sends a message of type m_1 to p_2, and immediately after p_3 sends a message of type m_2 to p_4. The opposite order, namely first the dispatching m_2 and only then m_1, is not allowed by the choreography. Assume that the choreography is *self-contained*, i.e. it does not allow other communication among its participants except what is explicitly modeled as message exchanges, and that the message exchanges are performed in *secrecy*, i.e. that only senders and recipients have access to the content of the message and know that the message exchange has occurred. Since p_3 does not partake the message exchange that delivers m_1, then p_3 has no means of knowing when it is expected of sending m_2. A choreography that is specified so that its participants cannot play their roles due to similar issues is said to be *unrealizable*.

Realizability is a fundamental property of interaction choreographies [MPR$^+$09]. In a sense, an unrealizable interaction choreography fails its purpose: it specifies a distributed messaging behavior that *cannot* be enacted accurately by its participants. In this work we present a method for the analysis of the realizability of interaction choreographies based on the concept of *participant awareness*. In a nutshell, the participant awareness is a symbolic representation of what a participant "knows" of the global state of the enactments it partakes. The realizability analysis presented in this work is

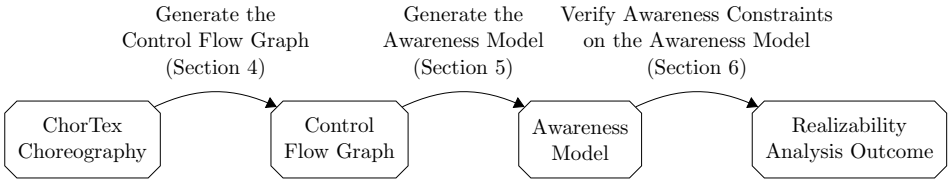


Figure 2: An outlook of the realizability analysis of ChorTex choreographies; the chamfered rectangles represent artifacts, and the arrows connecting them are steps of the realizability analysis.

specified on the basis of ChorTex, a choreography modeling language based on process algebras. We adopt ChorTex instead of an already existing modeling language to investigate the impact in terms of realizability of constructs that interrupt the enactment of others. In particular, ChorTex has exception throwing and handling constructs that are very similar to those available in orchestration languages like the Business Process Execution Language for Web Services (WS-BPEL), and that can be used to realize interrupting, event-based constructs like termination end events in BPMN v2.0. Our finding is that, due to the distributed nature of choreography enactments, modelers must use such constructs with *extreme* care.

Figure 2 provides an overview of the realizability analysis presented in this work. First, a Control Flow Graph (CFG) – a well-known data-structure often used in compilers theory [Lou97] – is generated from the choreography. Such CFG represents the events that occur in an enactment, e.g. the beginning or completion of an activity, and their ordering. In the second step, the nodes of the CFG are annotated with information on the *participant awareness*, which is a symbolic representation of which events can each participant observe, and that is calculated using techniques borrowed from the field of control-flow analysis in programming languages. A CFG whose nodes are annotated with the awareness of the participants is called Awareness Model (AWM). Finally, the realizability of the choreography is tested by verifying constraints on the awareness of the participants that, if verified, guarantee that the participants are able to play by their roles as specified.

This work is structured as follows. Section 2 presents ChorTex, the choreography modeling language that we employ in this work. In Section 3 we provide the necessary background on choreography realizability and state the definition of choreography realizability that we verify on ChorTex choreographies. Section 4 shows how to construct CFGs of ChorTex choreographies. Section 5 elaborates the concept of awareness and presents the algorithm to annotate nodes of a CFGs with the participant awareness, thus creating AWMs. Section 6 presents how the realizability constraints are specified and verified on AWMs. Finally, Section 7 concludes the work by discussing

the proposed method in light of the related work and our findings on the impact of interrupting constructs in terms of realizability.

2 ChorTex: A Choreography Modeling Language

ChorTex is based on Chor [YZCQ07], a choreography modeling language providing mechanisms for exception handling inspired by WS-BPEL. The syntax and operational semantic of ChorTex are presented in Section 2.1 and Section 2.2, respectively. The details and rationale of the differences between ChorTex over Chor are discussed in Section 2.3.

2.1 Syntax and Overview of ChorTex

The syntax of ChorTex is presented in Figure 3 by means of the Xtext Domain Specific Language (DSL)[1]. The syntax of the Xtext DSL is similar to 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 **Block** has at its two ends open and closed brackets, namely the '{' and '}' literals, which surround 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 Figure 3, the symbols +, ? and * represent cardinalities, meaning that the groups preceding them (groups are delimited by parentheses) have to appear at least one, zero or one times, or any number of times, respectively.

A choreography specifies a *body* which represents the "normal flow" of activities, and *exception handlers* which 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 the choreography are matched against the type of the exception that is thrown. 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*.

[1] Xtext (`www.eclipse.org/Xtext/`) is an 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: `http://www.eclipse.org/Xtext/documentation/`

Choreography: ‘chor’ (‘[’ name=**ID** ‘]’)? ‘(’ body=**Activity** (‘|’ namedExceptionHandlers+=**NamedExceptionHandler**)*
(‘|’ defaultExceptionHandler=**DefaultExceptionHandler**)? ‘)’;

NamedExceptionHandler: exceptionType=**ID** ‘:’ body=**BlockActivity**;

DefaultExceptionHandler: ‘∗’ ‘:’ body=**BlockActivity**;

Activity: **BasicActivity** | **ComplexActivity**;

BasicActivity: **SkipActivity** | **MessageExchangeActivity** | **OpaqueActivity** | **ThrowActivity**;

ComplexActivity: **BlockActivity** | **ChoiceActivity** | **IterationActivity** | **ParallelActivity** | **Choreography**;

SkipActivity: ‘skip’ (‘[’ name=**ID** ‘]’)?;

MessageExchangeActivity: (‘[’ name=**ID** ‘]’)? sender=**ID** ‘→’ messageType=**ID** ‘to’ recipients+=**ID** (‘,’ recipients+=**ID**)∗;

ThrowActivity: ‘throw’ (‘[’ name=**ID** ‘]’)? exceptionType=**ID**;

OpaqueActivity: ‘opaque’ (‘[’ name=**ID** ‘]’)? ‘(’ participants+=**ID** (‘,’ participants+=**ID**)+ ‘)’;

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

ChoiceActivity: ‘choice’ (‘[’ name=**ID** ‘]’)? decisionMaker=**ID** ‘either’ branches+=**BlockActivity** (‘or’ branches+=**BlockActivity**)+;

IterationActivity: ‘iteration’ (‘[’ name=**ID** ‘]’)? decisionMaker=**ID** ‘do’ body=**BlockActivity**;

ParallelActivity: ‘parallel’ (‘[’ name=**ID** ‘]’)? ‘do’ branches+=**BlockActivity** (‘and’ branches+=**BlockActivity**)+;

Figure 3: The syntax of ChorTex expressed using the Xtext DSL.

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, 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 exceptions propagating from it cannot be "resumed" after that the exception has been handled. Otherwise, if the enactment of body of the exception handler results in another exception been thrown, the choreography terminates 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.

The actual participants of a choreography, i.e. the entities such as services or individuals, are specified at design time in the choreography. We assume that each participant "knows" all the others, and that their identifiers are sufficient information for dispatching messages to them. Moreover, we assume that each participant is given "a copy" of the choreography, which is used as an artifact in the software development process of the participant implementation.

When the message exchange $p_s \rightarrow m$ to $p_{r_1}, \ldots, p_{r_n}$ is enacted, the participant p_s sends a message of type m to the participants $p_{r_1}, \ldots, p_{r_n}$. The participant that dispatches the message is called *sender*, i.e. p_s in the previous example. The participants that receive the message are called *recipients*. A participant cannot act as a sender and recipient in the same message exchange.[2] The type of the message that is exchanged over a message exchange is uniquely identified using an identifier like m. We assume that a participant can uniquely identify the type of a message by observing the latter's content. In the scope of Service Oriented Architecture (SOA) technologies, this is a realistic assumption: messages can include meta-data

[2]This assumption simplifies the realizability analysis presented in Section 6 without sacrificing the expressiveness of ChorTex. After all, a message sent by a participant to itself is more an *internal action* than a proper message exchange.

like information on their type, e.g. as SOAP headers. This assumption allows the recipients of a message to "trace back" which message exchange has been enacted solely on the basis of the message they have received.

In this work we do not consider synchronous message exchanging, as it is not realistic in the scope of Service-Based Applications (SBAs), i.e. distributed systems realized on the basis of SOA tenets. Instead, the message exchanges among the participants are asynchronous. We assume the asynchronous communications among the participants to have the following capabilities:

In-order reception: A participant receives messages in the same order in which they are sent;

Exactly-once reception: A message dispatched once by the sender is received exactly once by each of its recipients;

Always successful: No messages are lost or recipients are unreachable;

No message corruption: The sender and recipients of a message see exactly the same content;

These assumptions may sound strong; however, they are feasible in current state of the art of SOA through the adoption of technologies like WS-ReliableMessaging [FPD+09] (using "exactly-once" and "in-order" policies), and high-availability features of modern messaging services. It is not relevant in the scope of this work how asynchronous messaging is actually realized, e.g. if recipients have queues and which is their size. 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 exactly when the message reception and consumption by one recipient will occur. More specifically, there is no guarantee that multiple recipients of a message will consume it at the same time, nor in any particular order (e.g. the participant p_i before or after p_j).

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.

The activity `opaque` $(p_1, \ldots, p_n)$ represents an unspecified part of the choreography that involves the participants $p_1, \ldots, p_n$. 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. The particular message exchanges to be performed and their order can be specified (1) later in the modeling of the choreography, or (2) at run-time by the participants that partake that opaque activity in the fashion of ad-hoc modeling [AtHEvdA06, WRRM08]. 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 exception handling mechanisms in ChorTex are control-flow constructs for specifying the interruption of the enactment of concurrent activities, and the triggering of others as a result. Unlike programming languages like Java or orchestration languages like WS-BPEL, an exception thrown while enacting a ChorTex choreography is not represented by a data-structure. In a nutshell, 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.

The block activity $\{ A_1, \ldots, A_2 \}$ denotes the sequential enactment of the activities $A_1, \ldots, A_2$. 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` ... `and` A_n specifies the concurrent enactment of the *branches* $A_1, \ldots, A_n$. For simplicity, we assume each branch to be 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 been thrown, the other branches that have not yet completed are immediately terminated, the parallel activity is itself terminated, and the exception is propagated 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"). The decision about which of the $A_1 \ldots A_n$ activities, called *branches*, is enacted is taken *internally* by the participant p, which is said to be the *decision maker*. Since the decision is internal, the choice construct does not specify the criteria used by the decision maker for deciding which branch is executed. Is important to notice that there is no "visible" proof of the outcome of the decision maker's decision. The other participants 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 to iterate again the activity is taken internally by the decision maker p. If

an exception propagates from the body, the iteration activity is terminated and the exception is propagated to its parent.

2.2 Operational Semantics of ChorTex

Before detailing the operational semantics of ChorTex (Section 2.2.2), we need to lay some groundwork (Section 2.2.1).

2.2.1 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.

Definition 1 (Actions and Acting Participants)**.** The participants that perform a certain action are its *acting participants*. Table 1 correlates the various types of actions that are performed in ChorTex choreographies with their acting participants.[3]

Action	Acting Participants	Description
$[mex]\ p_s \xrightarrow{m} p_{r_1}, \ldots, p_{r_n}$	p_s	Dispatching of the message m by the sender p_s to the recipients $p_{r_1}, \ldots, p_{r_n}$ when enacting the message exchange activity mex
$[o]$ 🌫 $(p_1, \ldots, p_n)$	$p_1, \ldots, p_n$	Enactment of the opaque activity o by $p_1, \ldots, p_n$
$[c]\ p \overset{?}{\mapsto} \mathbf{x}, \mathbf{x} \in \{A_1, \ldots, A_n\}$	p	The decision maker p decides which of the branches $A_1, \ldots, A_n$ of the choice activity c must be enacted
$[i]\ p \overset{\circlearrowleft}{\mapsto} \mathbf{x}, \mathbf{x} \in \{\top, \bot\}$	p	The decision maker p decides whether to iterate the body of the iteration activity i; $\top$ and $\bot$ denote "true" and "false," respectively

Table 1: The types of actions and corresponding acting participants.

Definition 2 (Enactment Traces)**.** An *enactment trace* $(\!|\, \mathsf{a_1}, \ldots, \mathsf{a_n} \,|\!)$ is a sequence of actions $\mathsf{a_1}, \ldots, \mathsf{a_n}$ performed collectively by the participants during an enactment of a choreography. The concatenation of enactment traces is performed through the operator $\circ$, which is defined as follows:

$$(\!|\, \mathsf{a_1}, \ldots, \mathsf{a_m} \,|\!) \circ (\!|\, \mathsf{a'_1}, \ldots, \mathsf{a'_n} \,|\!) = (\!|\, \mathsf{a_1}, \ldots, \mathsf{a_m}, \mathsf{a'_1}, \ldots, \mathsf{a'_n} \,|\!)$$

The concatenation implies a temporal order between two traces that are united together. 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 (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.

[3] 🌫 is a weather symbol representing fog; it seems fitting, since "what happens" during the enactment of an opaque activity is not visible in the enactment trace.

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 Section 2.2.2. Notice that, is 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, can "kick-start" enactments of that choreography. (It is important to differentiate between initiating and non-initiating actions because the former are treated differently than the latter in terms of the realizability analysis, see Section 6.1.)

Definition 4 (Initiating Actions)**.** An action is *initiating* if its performing by the acting participants may cause the initiation of an enactment.

Intuitively, the possible initiating actions of a choreography are all those that can appear as first in a valid enactment trace. Notice that a choreography may have more than one initiating action. Consider, for example, the simple choreography shown in Figure 4. Since the activities that specify actions, namely the message exchange activities mex_1 and mex_2, are located in branches of the parallel activity, prl, both mex_1 and mex_2 can initiate an enactment of the choreography.

```
1 chor [chor1] (
2   [b] {
3     parallel [prl] do [b1] {
4       [mex1] p1 → m1 to p2
5     } and [b2] {
6       [mex2] p2 → m2 to p1
7     }
8   })
```

Figure 4: An example of choreography with multiple initiating actions.

This choreography requires *both* the message exchange activities mex_1 and mex_2 to be performed in order for an enactment to complete. Assume that at some point in time, first p_1 dispatches m_1 to p_2, and then p_2 dispatches m_2 to p_1. But does the dispatching of m_2 by p_2 represent the initiation of a new enactment, or just the completion of the one initiated by p_1 with the dispatching of m_1? To solve this dilemma, we assume that participants employ a *session* mechanism that allows them correlate message exchanges (but, more generally, action) to enactments. Fundamentally, each enactment is uniquely identified by an identifier generated upon the performing of the initiating action, and that is later included in the meta-data of each message exchanged between the participants. Similar techniques are used for Hypertext Transfer

Protocol (HTTP) session tracking and in distributed transaction management. In the current SOA landscape, such session mechanism is easily realized by means of, for example, WS-Addressing [BCC+06].

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 activity specified by the choreography (see the activity life-cycles in presented in Section 2.2.2), and the *enactment mode*, i.e. whether if an exception is been propagated or, on the contrary, the enactment is "proceeding normally."

Definition 5 (Enactment States)**.** The state χ the enactment of a choreography is a tuple:

$$\chi := [\![\delta, \sigma, \mu]\!]$$

The symbol δ denotes the *enactment environment*, i.e. a *key-value map* of the states of the activities specified by the choreography. The keys of the enactment environment are the activities specified by the choreography, and the respective values are the activities' own states (specified later in Section 2.2.2). The fact that the activity A is in the state s is denoted by:

$$\delta[A] = s$$

The symbol σ represents the enactment trace. Finally, μ denotes the *enactment mode*, i.e. if the enactment is being enacted "normally," denoted by ✓, or else an exception e is been propagated, denoted by $\lightning\, e$.

2.2.2 Operational Semantics of ChorTex Activities

This section presents the operational semantics of ChorTex activities as both State Diagrams (SDs) of ChorTex elements (Figure 5 through Figure 13), as well as structured operational semantics (Figure 14 and Figure 15). 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. SDs of the life-cycle of ChorTex elements, on the other hand, are easier to read and provide a perspective on the enactment of a ChorTex choreography focused on the single choreography elements.

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 14 and Figure 15 adopts a "state-based" style, representing the current progress of the enactment as a combination of the states of the single activities. The reason for the adoption of this style is that the resulting operational semantics mirrors closely the life-cycles of the activities that, in our opinion, are far more intuitive to the reader familiar with workflows and service composition languages than "mainstream" structured operational semantics.

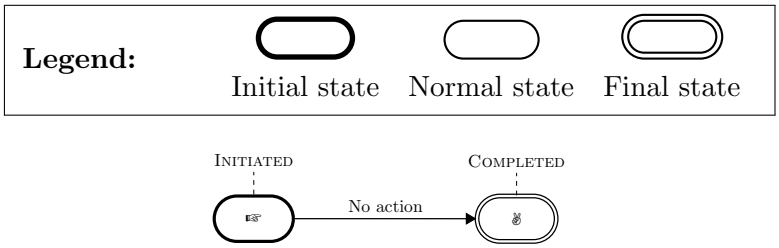


Figure 5: The enactment life-cycle of a skip activity.

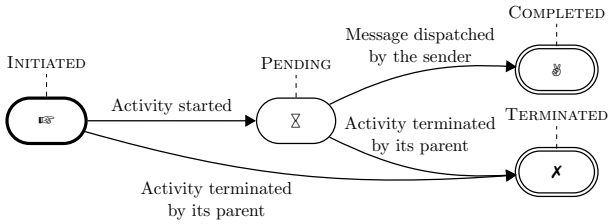


Figure 6: The enactment life-cycle of a message exchange activity.

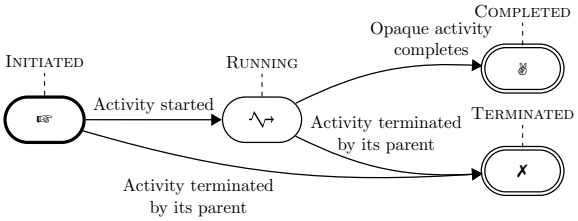


Figure 7: The enactment life-cycle of an opaque activity.

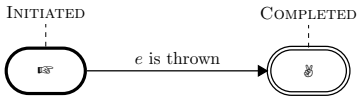


Figure 8: The enactment life-cycle of a throw activity.

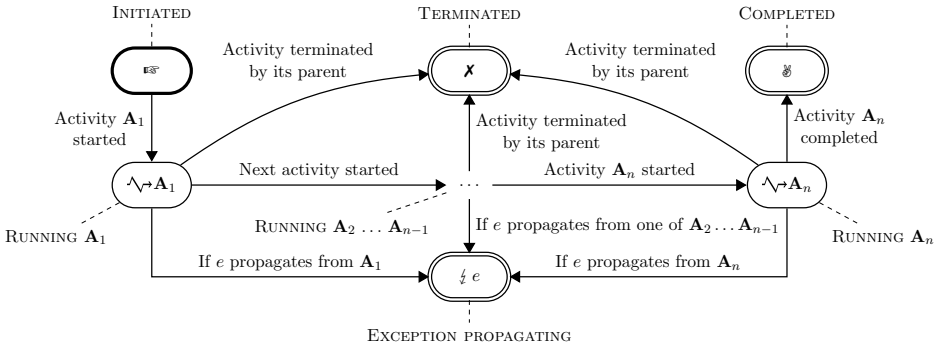


Figure 9: The enactment life-cycle of a block.

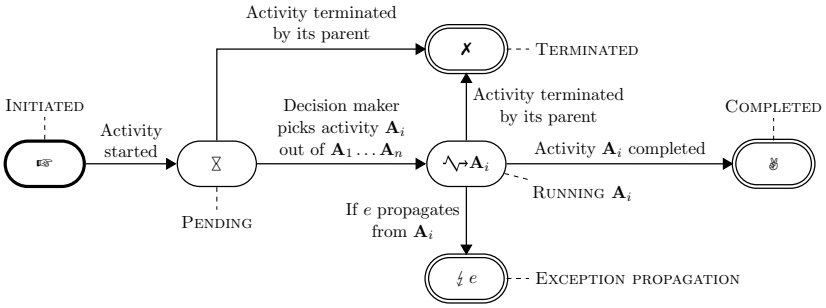


Figure 10: The enactment life-cycle of a choice activity.

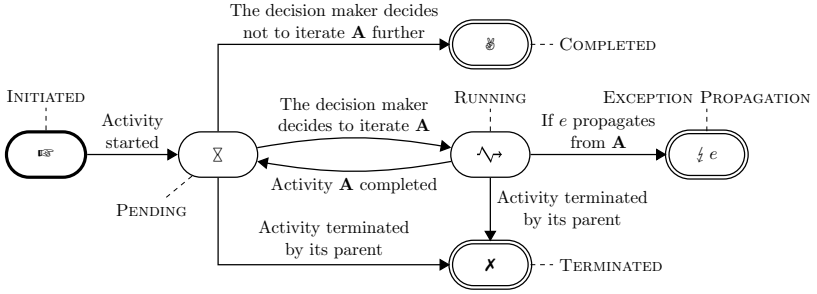


Figure 11: The enactment life-cycle of an iteration activity.

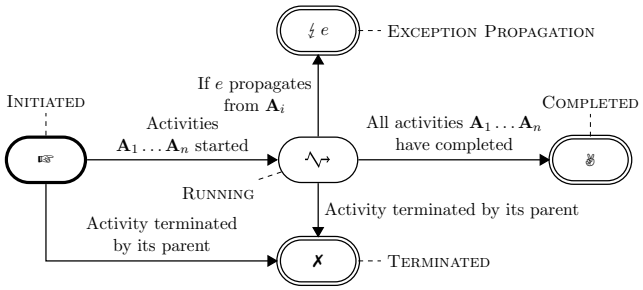


Figure 12: The enactment life-cycle of a parallel activity.

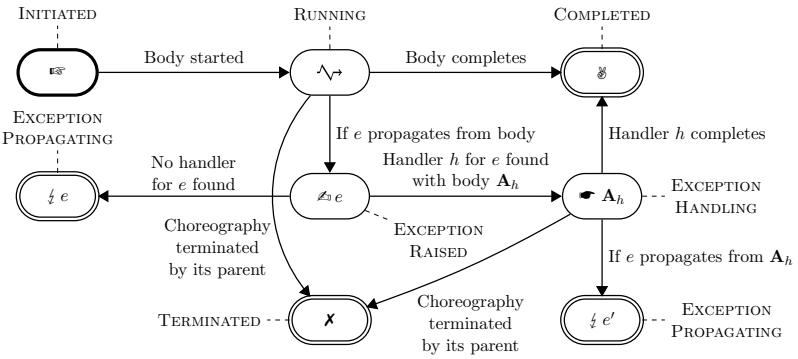


Figure 13: The enactment life-cycle of a choreography.

In Figure 14 and Figure 15, 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:

$$\delta[A \backslash s]$$

The symbols that represent the states of the activities are the same adopted in the activities' life-cycles depicted in Figure 5 through Figure 13.

A skip activity, the life-cycle of which is shown in Figure 5, is instantaneous (see Section 2.1). 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 is ☞), it completes successfully (i.e. its state becomes ✌)." Notice that skip activities can be enacted only when the enactment mode is "normal" (denoted by ✓). However, since skip activities are instantaneous, when they are initiated can never be interrupted by exception propagation.

The life-cycle of message exchange activities is shown in Figure 6. 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 ⧖ (Rule Message Exchange Initiation). In "normal" enactment mode, the message exchange activity performs the transition from the Running to the Completed state when the sender dispatches the message to the recipients (Rule Message Exchange Completion). Notice that, since the (Rule Message Exchange Completion) assumes the enactment mode to be "normal," participants that dispatch messages when the enactment state in "exception propagation" mode ($\lightning\, e$) are violating the choreography. 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. Since message exchange activities are not instantaneous, they may be terminated while they are in the state Pending. 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 – i.e. the changing of its state to Terminated, denoted by ✗ – 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). The

In the reminder, s denotes: *skip*

$$\frac{\delta[s] = \text{☞}}{[\![\delta, \sigma, \checkmark]\!] \to [\![\delta[s \backslash \text{✌}], \sigma, \checkmark]\!]} \quad \text{(Skip)}$$

In the reminder, mex denotes: $p_s \to m$ to $p_{r_1}, \ldots, p_{r_n}$

$$\frac{\delta[mex] = \text{☞}}{[\![\delta, \sigma, \checkmark]\!] \to [\![\delta[mex \backslash \text{⧖}], \sigma, \checkmark]\!]} \quad \text{(Message Exchange Initiation)}$$

$$\frac{\delta[mex] = \text{⧖} \wedge p_s \text{ dispatches } m \text{ to } p_{r_1}, \ldots, p_{r_n}}{[\![\delta, \sigma, \checkmark]\!] \to [\![\delta[mex \backslash \text{✌}], \sigma \circ (\!| [mex]\ p_s \xrightarrow{m} p_{r_1}, \ldots, p_{r_n} |\!), \checkmark]\!]} \quad \text{(Message Exchange Completion)}$$

In the reminder, o denotes: opaque $(p_1, \ldots, p_n)$

$$\frac{\delta[o] = \text{☞} \wedge p_1, \ldots, p_n \text{ start enacting } o}{[\![\delta, \sigma, \checkmark]\!] \to [\![\delta[o \backslash \text{↯}], \sigma, \checkmark]\!]} \quad \text{(Opaque Initiation)}$$

$$\frac{\delta[o] = \text{↯} \wedge p_1, \ldots, p_n \text{ complete enacting } o}{[\![\delta, \sigma, \checkmark]\!] \to [\![\delta[o \backslash \text{✌}], \sigma \circ (\!| [o] \text{ ▦ } (p_1, \ldots, p_n) |\!), \checkmark]\!]} \quad \text{(Opaque Completion)}$$

In the reminder, t denotes: throw e

$$\frac{\delta[t] = \text{☞}}{[\![\delta, \sigma, \checkmark]\!] \to [\![\delta[t \backslash \text{✌}], \sigma, \lightning\ e]\!]} \quad \text{(Throw)}$$

In the reminder, b denotes: $\{ A_1; A_2; \ldots; A_n \}$

$$\frac{\delta[b] = \text{☞}}{[\![\delta, \sigma, \checkmark]\!] \to [\![\delta[b \backslash \text{↯} A_1][A_1 \backslash \text{☞}], \sigma, \checkmark]\!]} \quad \text{(Block Initiation)}$$

$$\frac{\delta[b] = \text{↯} A_i \wedge \delta[A_i] = \text{✌} \wedge i < n}{[\![\delta, \sigma, \checkmark]\!] \to [\![\delta[b \backslash \text{↯} A_{i+1}][A_{i+1} \backslash \text{☞}], \sigma, \checkmark]\!]} \quad \text{(Block Next Activity Initiation)}$$

$$\frac{\delta[b] = \text{↯} A_n \wedge \delta[A_n] = \text{✌}}{[\![\delta, \sigma, \checkmark]\!] \to [\![\delta[b \backslash \text{✌}], \sigma, \checkmark]\!]} \quad \text{(Block Completion)}$$

$$\frac{\delta[b] = \text{↯} A_i \wedge \delta[A_i] = \lightning\ e}{[\![\delta, \sigma, \lightning\ e]\!] \to [\![\delta[b \backslash \lightning\ e], \sigma, \lightning\ e]\!]} \quad \text{(Block Exception)}$$

$$\frac{\delta[b] = \text{✗}}{[\![\delta, \sigma, \lightning\ e]\!] \to [\![\delta[A_1 \backslash \text{✗}] \ldots [A_n \backslash \text{✗}], \sigma, \lightning\ e]\!]} \quad \text{(Block Termination)}$$

In the reminder, c denotes: choice p either A_1 or $\ldots$ or A_n

$$\frac{\delta[c] = \text{☞}}{[\![\delta, \sigma, \checkmark]\!] \to [\![\delta[c \backslash \text{⧖}], \sigma, \checkmark]\!]} \quad \text{(Choice Initiation)}$$

$$\frac{\delta[c] = \text{⧖} \wedge p \text{ picks } A_i \text{ out of } A_1, \ldots, A_n}{[\![\delta, \sigma, \checkmark]\!] \to [\![\delta[c \backslash \text{↯} A_i][A_i \backslash \text{☞}], \sigma \circ (\!| [c]\ p \overset{?}{\mapsto} \mathbf{A}_i |\!), \checkmark]\!]} \quad \text{(Choice Decision)}$$

$$\frac{\delta[c] = \text{↯} A_i \wedge \delta[A_{i \in [1,n]}] = \text{✌}}{[\![\delta, \sigma, \checkmark]\!] \to [\![\delta[c \backslash \text{✌}], \sigma, \checkmark]\!]} \quad \text{(Choice Completion)}$$

$$\frac{\delta[c] = \text{↯} A_i \wedge \delta[A_{i \in [1,n]}] = \lightning\ e}{[\![\delta, \sigma, \lightning\ e]\!] \to [\![\delta[c \backslash \lightning\ e], \sigma, \lightning\ e]\!]} \quad \text{(Choice Exception)}$$

$$\frac{\delta[c] = \text{✗} \wedge \delta[A_{i \in [1,n]}] \neq \text{✌}}{[\![\delta, \sigma, \lightning\ e]\!] \to [\![\delta[A_i \backslash \text{✗}], \sigma, \lightning\ e]\!]} \quad \text{(Choice Termination)}$$

Figure 14: The structured operational semantics of ChorTex (Part 1).

In the reminder, i denotes: iteration p do A

$$\frac{\delta[i] = \text{☞}}{[\![\delta, \sigma, \checkmark]\!] \to [\![\delta[i \backslash \text{⧖}], \sigma, \checkmark]\!]} \quad \text{(Iteration Initiation)}$$

$$\frac{\delta[i] = \text{⧖} \wedge p \text{ internally decides to iterate A}}{[\![\delta, \sigma, \checkmark]\!] \to [\![\delta[i \backslash \text{↯}][A \backslash \text{☞}], \sigma \circ (\!| [i]\ p \overset{\circlearrowleft}{\mapsto} \top |\!), \checkmark]\!]} \quad \text{(Iteration Decision True)}$$

$$\frac{\delta[i] = \text{⧖} \wedge p \text{ internally decides not to iterate A}}{[\![\delta, \sigma, \checkmark]\!] \to [\![\delta[i \backslash \text{😴}], \sigma \circ (\!| [i]\ p \overset{\circlearrowleft}{\mapsto} \bot |\!), \checkmark]\!]} \quad \text{(Iteration Decision False)}$$

$$\frac{\delta[i] = \text{↯} \wedge \delta[A] = \text{😴}}{[\![\delta, \sigma, \checkmark]\!] \to [\![\delta[i \backslash \text{⧖}], \sigma, \checkmark]\!]} \quad \text{(Iteration Body Completion)}$$

$$\frac{\delta[i] = \text{✗} \wedge \delta[A] \neq \text{😴}}{[\![\delta, \sigma, \lightning\ e]\!] \to [\![\delta[A \backslash \text{✗}], \sigma, \lightning\ e]\!]} \quad \text{(Iteration Termination)}$$

In the reminder, prl denotes: parallel do A_1 and $\ldots$ and A_n

$$\frac{\delta[prl] = \text{☞}}{[\![\delta, \sigma, \checkmark]\!] \to [\![\delta[prl \backslash \text{↯}][A_1 \backslash \text{☞}]\ \ldots\ [A_n \backslash \text{☞}], \sigma, \checkmark]\!]} \quad \text{(Parallel Initiation)}$$

$$\frac{\delta[prl] = \text{↯} \wedge \forall i \in [1,n] : \delta[A_i] = \text{😴}}{[\![\delta, \sigma, \checkmark]\!] \to [\![\delta[prl \backslash \text{😴}], \sigma, \checkmark]\!]} \quad \text{(Parallel Completion)}$$

$$\frac{\delta[prl] = \text{↯} \wedge \delta[A_{i \in [1,n]}] = \lightning\ e \wedge \{A_{j \in [1,n]} \mid \delta[A_j] \neq \text{😴}\} = \{A'_1, \ldots, A'_m\}}{[\![\delta, \sigma, \lightning\ e]\!] \to [\![\delta[prl \backslash \lightning\ e][A'_1 \backslash \text{✗}] \ldots [A'_m \backslash \text{✗}], \sigma, \lightning\ e]\!]} \quad \text{(Parallel Exception)}$$

$$\frac{\delta[prl] = \text{✗} \wedge \{A_{i \in [1,n]} \mid \delta[A_i] \neq \text{😴}\} = \{A'_1, \ldots, A'_m\}}{[\![\delta, \sigma, \lightning\ e]\!] \to [\![\delta[A'_1 \backslash \text{✗}] \ldots \delta[A'_m \backslash \text{✗}], \sigma, \lightning\ e]\!]} \quad \text{(Parallel Termination)}$$

In the reminder, $chor$ denotes: chor $(A \mid e_1 : A_1 \mid \ldots \mid e_n : A_n \mid * : A_*)$

$$\frac{\delta[chor] = \text{☞}}{[\![\delta, \sigma, \checkmark]\!] \to [\![\delta[chor \backslash \text{↯} A][A \backslash \text{☞}], \sigma, \checkmark]\!]} \quad \text{(Choreography Initiation)}$$

$$\frac{\delta[chor] = \text{↯} A \wedge \delta[A] = \text{😴}}{[\![\delta, \sigma, \checkmark]\!] \to [\![\delta[chor \backslash \text{😴}], \sigma, \checkmark]\!]} \quad \text{(Choreography Completion)}$$

$$\frac{\delta[chor] = \text{↯} A \wedge \delta[A] = \lightning\ e}{[\![\delta, \sigma, \lightning\ e]\!] \to [\![\delta[chor \backslash \text{✍} e], \sigma, \checkmark]\!]} \quad \text{(Choreography Exception 1)}$$

$$\frac{\delta[chor] = \text{✍} e \wedge \exists\, i \in [1,n] : e_i = e \wedge A_i \text{ is body of the named exception handler for } e_i}{[\![\delta, \sigma, \lightning\ e]\!] \to [\![\delta[chor \backslash \text{☛} A_i][A_i \backslash \text{☞}], \sigma, \checkmark]\!]} \quad \text{(Choreography Exception 2)}$$

$$\frac{\delta[chor] = \text{✍} e \wedge \nexists\, i \in [1,n] : e_i = e \wedge A_* \text{ is body of the default exception handler}}{[\![\delta, \sigma, \lightning\ e]\!] \to [\![\delta[chor \backslash \text{☛} A_*][A_* \backslash \text{☞}], \sigma, \checkmark]\!]} \quad \text{(Choreography Exception 3)}$$

$$\frac{\delta[chor] = \text{✍} e \wedge \nexists\, i \in [1,n] : e_i = e \wedge \text{ no default exception handler specified}}{[\![\delta, \sigma, \lightning\ e]\!] \to [\![\delta[chor \backslash \lightning\ e], \sigma, \lightning\ e]\!]} \quad \text{(Choreography Exception 4)}$$

$$\frac{\delta[chor] = \text{☛} A_h \wedge \delta[A_h] = \text{😴}}{[\![\delta, \sigma, \checkmark]\!] \to [\![\delta[chor \backslash \text{😴}], \sigma, \checkmark]\!]} \quad \text{(Choreography Exception Handling 1)}$$

$$\frac{\delta[chor] = \text{☛} A_h \wedge \delta[A_h] = \lightning\ e}{[\![\delta, \sigma, \lightning\ e]\!] \to [\![\delta[chor \backslash \lightning\ e], \sigma, \lightning\ e]\!]} \quad \text{(Choreography Exception Handling 2)}$$

$$\frac{\delta[chor] = \text{✗} \wedge \delta[A] = \text{↯}}{[\![\delta, \sigma, \lightning\ e]\!] \to [\![\delta[A \backslash \text{✗}], \sigma, \lightning\ e]\!]} \quad \text{(Choreography Termination 1)}$$

$$\frac{\delta[chor] = \text{✗} \wedge \delta[A_{i \in [1,n]}] = \text{↯}}{[\![\delta, \sigma, \lightning\ e]\!] \to [\![\delta[A_i \backslash \text{✗}], \sigma, \lightning\ e]\!]} \quad \text{(Choreography Termination 2)}$$

$$\frac{\delta[chor] = \text{✗} \wedge \delta[A_*] = \text{↯}}{[\![\delta, \sigma, \lightning\ e]\!] \to [\![\delta[A_* \backslash \text{✗}], \sigma, \lightning\ e]\!]} \quad \text{(Choreography Termination 3)}$$

Figure 15: The structured operational semantics of ChorTex (Part 2).

enactment of a message exchange activity cannot cause the throwing of an exception.

An initiated opaque activity changes its state 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 its 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 been changed to TERMINATED, denoted by ✗ – occurs while executing a rule that processes the termination of the complex activity that is the parent of that opaque activity. The enactment of an opaque activity cannot cause the throwing of an exception.

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

When a block is initiated, it initiates "on cascade" its first nested activity (Rule BLOCK INITIATION). A block can be initiated only when the enactment mode is "normal" (Rule BLOCK INITIATION). In a "normal" enactment mode, the completion of one nested activity that is not the last in the block triggers the initiation of the following nested activity (Rule BLOCK NEXT ACTIVITY INITIATION). In a "normal" enactment mode, the completion of the last nested activity results in the completion of the block (Rule BLOCK COMPLETION). If the enactment of one of the nested activities of a block results in an exception e propagating, the block is terminated with the state $\lightning\, e$, i.e. the exception propagates to the block's parent activity (Rule BLOCK EXCEPTION). Finally, when a block is terminated by its parent activity, i.e. its state changes to ✗, the nested activity currently running is also terminated (Rule BLOCK TERMINATION). The states of the previously completed nested activities 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 mode is $\lightning\, e$.

Choice activities, once initiated, enter the state PENDING (Rule CHOICE INITIATION), which represents the "waiting" for the internal decision performed by the decision maker about which of the branches to enact. 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 selected branch results in an exception propagating, 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 (Rule Choice Termination). The evaluation of the decision by the decision maker cannot cause an exception being thrown.

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 iterate the body (Rule Iteration Decision False). 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). Similarly to the case of choice activities, evaluation of the decision by the decision maker cannot cause an exception being thrown.

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 an exception propagating, 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).

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). On the contrary, if the enactment of the body results in the propagation of an exception of type e, the state of the choreography changes from Running

to Exception Raised (Rule Choreography Exception 1). When the choreography enters the state Exception Raised, the exception handlers defined by the choreography are matched against the exception type e. If a named exception handler matches the type of the propagating exception, its body is initiated (Rule Choreography Exception 2) and the state of the choreography changes to Exception Handling. 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 no matching named exception handler is found and the choreography defines no 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 was 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 an exception of type e', the choreography's state transitions to Exception Propagation (denoted by $\lightning\, e'$), and the exception is propagated to the choreography's parent (Rule Choreography Exception Handling 2). 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.

2.3 Differences between Chor and ChorTex

At first sight, ChorTex adopts a more natural language-like syntax than its predecessor Chor [YZCQ07]. However, differences between Chor and ChorTex go deeper than just the syntax, and are treated in the reminder.

2.3.1 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.) The execution of Chor's basic activities is instantaneous, infallible (i.e. it never results in an exception being thrown), 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 [YZCQ07]; presumably, basic activities are included in Chor because of its "twin" orchestration language Role. In fact, during the process of projection, a basic activity a specified in the choreography is transformed into a Role activity in the role of the participant that executes a.

In our opinion, allowing the specification of internal activities in a choreography language focusing on interaction modeling is a 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 – i.e. *private* – to the participants that execute them. Unlike the choice and iteration constructs, whose internal activities (the decisions) 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 must be enacted by multiple participants.

2.3.2 Choreography Nesting vs. Choreography Referencing

In Chor, choreographies are uniquely identified by name and can be referenced from inside other choreographies using the `perf` construct. When the `perf` C_m activity is traversed, the choreography identified by C_m is enacted. The completion of C_m leads to the completion of `perf` C_m. Similarly, the termination of C_m because of the propagation of an uncaught exception results in that exception propagating outside `perf` C_m. In ChorTex we replace choreography referencing with choreography nesting. Choreography referencing simplifies the modular definition of choreographies; however, it also complicates considerably the exposition of the realizability analysis presented in Section 5 by requiring the adoption of inter-procedural flow analysis techniques. Our design decision is taken for reasons of understandability of the exposition. Nevertheless, it is straightforward to extend the realizability analysis presented in Section 5 to support choreography referencing in ChorTex by means of the inter-procedural flow analysis techniques available in the state of the art, e.g. [Mye81, Cal88, RHS95, MO04].

2.3.3 Finalization Handlers

An extension to Chor presented in [YZCQ07] 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. In fact, presenting the generation and analysis of CFGs that

involve finalization handlers is much more convoluted than without it. Known techniques of CFG analysis such as [SH00] can deal with finalization handling, and can be adapted to deal with ChorTex choreographies with finalization handlers.

3 Choreography Realizability

The present section provides the necessary background on choreography realizability (Section 3.1) and specifies which type of realizability for ChorTex choreographies we are going to analyze in the reminder of this work (Section 3.2).

3.1 Aspects of Choreography Realizability

The state of the art is rich with heterogeneous definitions of choreography realizability, see e.g. [AEY03,AEY05,FBS05,KP06,SBFZ07,MFEH07,BFS07, BF08, BH10, HB10]. A very general – and intentionally under-specified – definition of choreography realizability is the following.

Definition 6 (Generic Definition of Choreography Realizability)**.** A choreography is *realizable* if it is possible to devise participant implementations for its roles that, when interacting with each other, are *behaviorally equivalent* to the choreography.

Definition 6 is under-specified because it does not clarify *which of the many possible behavioral-equivalence relations* is required between the composition of the participant implementations and the choreography. The adopted notion of behavioral equivalence, in turn, depends to some extent on design decisions and assumptions that underpin each choreography modeling language. Naturally, this has lead to a variety of different definitions of choreography realizability in the state of the art which can be classified according to the following three dimensions proposed in [Dec09]:

Complete vs. subset of the behavior: How *much* of the behavior of the choreography can be enacted by the composition of the participants implementations. A definition of choreography realizability requires *complete* behavior if all and only the behaviors (e.g. traces) that are specified by the choreography can be enacted by the composition of the participant implementations. Such definitions of realizability are often called *strong*, see e.g. [MFEH07, BF07, RS11]. If not all the choreography's behaviors must necessarily be enactable by the composition of the participant implementations, the definition of choreography realizability requires a *subset of the behavior*. Such definitions are usually labeled as "weak," see e.g. [AEY05, SBFZ07].

Communication model: What are the assumptions on functional and non-functional characteristics of the communication channels connecting the participants, e.g. synchronous, asynchronous with queues of fixed size, asynchronous of queues of infinite size, and whether the communication channels preserve the ordering of the messages in the queues.

Equivalence notion: The "strength" of the required equivalence notion, e.g. trace-equivalence, language-equivalence or bi-simulation [vG93, BIM95].

Given the dimensions listed above, it is clear that there is no "one-size-fits-all" definition of choreography realizability. Particularly interesting is the relevance of the communication model. To be formally specified, a choreography modeling language must necessarily make assumptions on how the participants communicate with each other. In other words, the communication model is a *design decision* for choreography modeling languages. This suggests that definitions of choreography realizability are necessarily "tailored" to some extent to specific choreography modeling languages.[4] The following section presents the definition of realizability for ChorTex choreographies we investigate in this work.

3.2 Strong Realizability of ChorTex Choreographies

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

Definition 7 (Conversations)**.** The *conversation* c_{en} performed during an enactment is the enactment trace σ restricted to only the actions that describe interactions among participants, i.e. message exchanges and opaque activities (which *may* involve interactions among the participants).

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

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

In this work we adapt to ChorTex choreographies the definition of *strong realizability* proposed in [KP06]:

[4] Actually, this coupling between choreography modeling languages and definitions of choreography realizability helps explaining the large variety of (and the limited comparison among) of the latter in the state of the art.

Definition 8 (Strong Realizability of ChorTex choreographies)**.** A ChorTex choreography is strongly realizable if exist participant implementations for its roles the composition of which is language-equivalent in terms of conversations to the original choreography.

In other words, the definition of strong realizability requires that it is possible to create participant implementations that, during every possible enactment, *preserve the ordering of the interactions between the participants* as specified by the choreography.

In terms of the three dimensions of choreography realizability discussed in Section 3.1, the notion of strong realizability we adopt in this work is classified as follows:

Completeness of the behavior: Complete.

Communication model: Asynchronous, in-order delivery with queues of infinite length (see Section 2);

Equivalence notion: Language-equivalence of conversations.

The rationale for choosing strong realizability as defined in Definition 8 over others is rooted in the fact that choreographies are often used as technical contracts among the participants (see Section 1). Every conversation intensionally defined by a strongly realizable choreography is enactable by the participants, but no restrictions are put on their internal actions. Also very important is what the strong realizability *does not* require, and in particular: (1) preservation of the ordering of reception and consumption of messages by the recipients and (2) preservation of the ordering of the internal activities, namely the decisions by the decision makers of choice and interaction activities. Recall that ChorTex assumes asynchronous messaging. One of the implications of the asynchronous messaging model assumed is that there are no guarantees about "when" a recipient will consume a message it has received (see Section 2.1). Therefore, there is no way to enforce a particular order between consumption of messages by recipients and the actions in the conversation. (An exception is when a recipient is required to take an action in response of a message it has consumed; in this case, the ordering is "implicitly" enforced, because due to how the choreography is specified, the recipient will not perform the action until it has consumed the message.) The ordering of internal actions performed by the participants cannot likewise be enforced. For example, nothing prohibits a participant to decide in advance, possibly even before an enactment begins, which decisions it will take. In a sense, this is equivalent to repeatedly throwing a dice before the beginning of a game of "Sorry!"[5] recording in order the outcomes, and

[5]Sorry! is a 1929 board game in which the players need to travel across the board with all their pieces faster than their opponents. The distance covered by one piece over a move

then using them one in the order they were thrown whenever required in the game.

The trade-off of strong realizability between what is required and what is not is clearly ideal for technical contracts such as a choreographies: the participants can provide implementations that comply with all the obligations (i.e. they are able to perform all the specified behaviors), and no other limitations are posed on how each participant realizes its own role.

4 From Choreographies to Control Flow Graphs

The realizability analysis for ChorTex choreographies presented in this work builds on top of the flow analysis framework (see e.g. [All70, Aho07]). The flow analysis framework is used to study *static properties* of computer programs, i.e. properties that can be inferred from the structure of the programs at design/compilation-time such as dominator and post-dominator analysis, reaching definitions and data dependencies [Aho07]. The flow analysis framework is based on the concept of Control Flow Graph (CFG). The CFG of a program (e.g. in Java) is a directed graph in which each represents one of the program's instructions, and an edge connecting two nodes represents a control dependency between the respective instructions. Section 4.1 explains how to construct CFGs from ChorTex choreographies, while Section 4.2 correlates the CFGs so constructed and the operational semantics of ChorTex presented in Section 2.2.

4.1 Construct CFGs of ChorTex Choreographies

We adapt to ChorTex choreographies the approach proposed by [SH00] for constructing CFGs of Java programs, thus exploiting the similarities between the exception handling mechanisms of ChorTex and those of general-purpose programming languages. Table 2 and Table 3 show how to construct CFGs of ChorTex choreographies by mapping every choreography activity to a CFG sub-graph. The CFG nodes with the gray background in Table 2 and Table 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 labeled as "first" in the sub-graph resulting from the activity **A**. Similarly, the condition "from last (**A**)" means that

depends on the result of a dice throw. When a piece of a player reaches a location on the board that is already occupied by someone else's piece, the latter is moved back to its owner's starting zone; hence the title of the game. (However, in the authors' experience, such a proclamation of discontentment by a player that sent back someone's piece is virtually always a lie intended to further vex the opponent.) A scan of the rules of "Sorry!" is available on http://www.hasbro.com/common/instruct/sorry.pdf

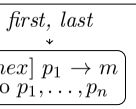
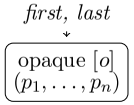
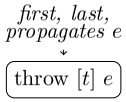
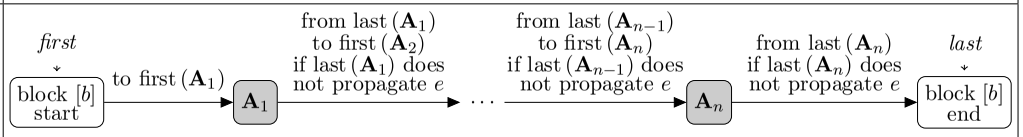
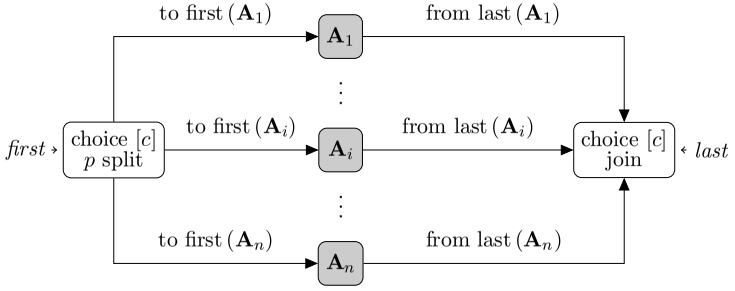
ChorTex Activity	**Corresponding Control Flow Graph**
skip $[s]$	
$[mex]\ p_1 \to m$ to $p_1, \ldots, p_n$	
opaque $[o]$ $(p_1, \ldots, p_n)$	
throw $[t]$ e	
$[b]$ { $\mathbf{A}_1$; … ; $\mathbf{A}_n$ }	
choice $[c]$ p either $\mathbf{A}_1$ or … or $\mathbf{A}_n$	

Table 2: The mapping from ChorTex activities to Control Flow Graphs (Part 1).

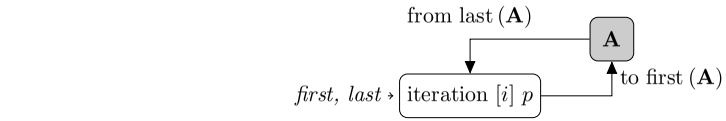
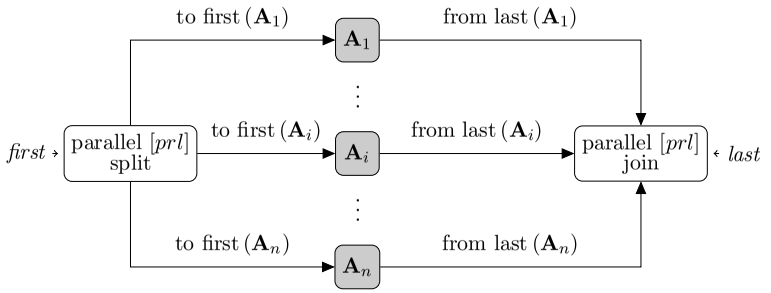
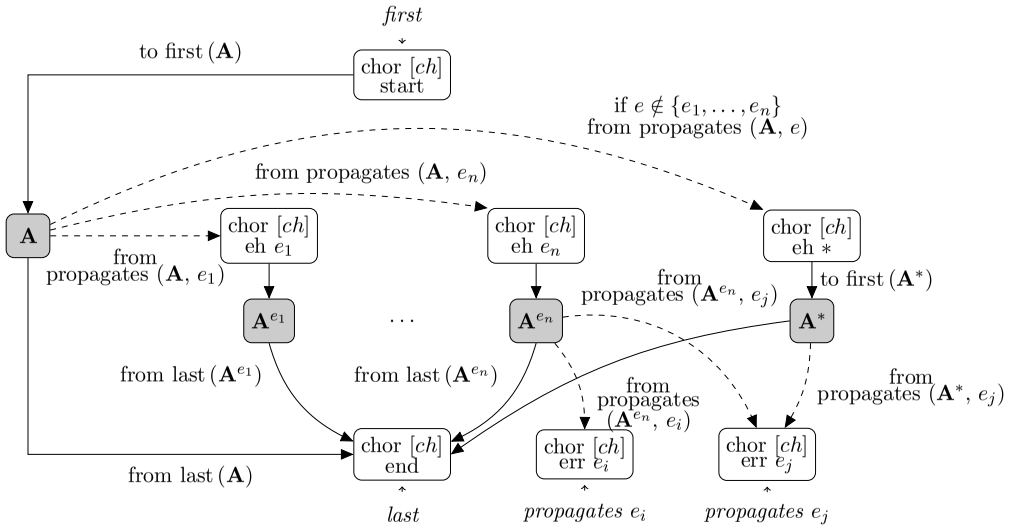
ChorTex Activity	**Corresponding Control Flow Graph**
iteration $[i]$ p do $\mathbf{A}$	
parallel $[prl]$ do $\mathbf{A}_1$ and ... and $\mathbf{A}_n$	
chor $[ch]$ ($\mathbf{A}$ | $e_1 : \mathbf{A}^{e_1}$ | ... | $e_n : \mathbf{A}^{e_n}$ | $* : \mathbf{A}^*$)	

Table 3: The mapping from ChorTex activities to Control Flow Graphs (Part 2).

the source of the labeled edge is the node labeled as "last" in the sub-graph resulting from the activity **A**. It should be noted that, due to the definition of "first" and "last" in Table 2 and Table 3, each CFG sub-graph has precisely one "first" and one "last" node.

Each node of the CFG generated from a ChorTex choreography represents the *firing* of exactly one event, e.g. the beginning or completion of an activity, during an enactment of the choreography. Consequently, the CFG edges represent the order in which the firing of events may occur while enacting 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 4.2, as it requires an understanding of how enacting activities relates to the firing of events.

The CFG nodes `block [`b`] start` and `block [`b`] end` that are generated from a block activity named b represent the beginning and completion of the enactment of b, respectively. Similarly, `choice [`c`]` p `split` and `choice [`c`] join` represent the beginning and completion of the choice activity c, respectively.[6]

The node `iteration [`i`]` p represents at the same time (1) beginning, (2) completion and (3) the taking of the decision by p of the iteration activity i.[7] 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 (represented by the node `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 labeled "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 CFG must not have a control-flow edge connecting $\mathbf{A}_1$ with $\mathbf{A}_2$.[8] 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 2 and Table 3. It should be noted that only the rule for

[6] It 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.

[7] It would be possible to represent separately these events with distinct nodes, but it would not provide any concrete advantage in terms of control-flow analysis.

[8] In ChorTex it is indeed possible to specify a block { `throw` e`; skip` }. The skip, in this case, is "dead-code," which is usually considered something to be avoided, but not a defect in itself. However, such cases may be excluded by dedicated well-formedness rules, e.g. that only the last activity in a block can be a throw activity.

creating CFG 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.

Figure 16 introduces the running example of this work, namely a choreography that specifies sub-choreographies and has a 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, which terminates the choreography. The exception propagation is evident in Figure 17, which shows the CFG resulting from the choreography in Figure 16. It is interesting to notice that there are some nodes, namely `chor [`$chor_1$`] eh` $*$, `skip [`$skip_2$`]`, `chor [`$chor_2$`] eh` $*$ and `skip [`$skip_1$`]`, that are not reachable by traversing the control flow edges from the start node of the CFG, i.e. the "first" node of the sub-graph generated from the root choreography. In Figure 17, 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 [`$choice_1 either$`] 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 [`$choice_1 either$`] end` nodes. The other unreachable nodes, instead, represent exception handlers that are never triggered.

4.2 Reconciling Actions and Events in ChorTex Enactments

On the basis of method to construct CFGs of ChorTex choreographies presented in Section 4.1, this section (1) provides a formal interpretation of the ordering of events as specified by the edges in the CFGs and (2) relate the events with the actions performed by participants that constitute an enactment trace (see Section 2.2). The content of this section is instrumental to the definition of the concept of awareness and the constraints for verifying the strong realizability of ChorTex choreographies that is presented later in Section 5 and Section 6. Figure 18 summarizes the terminology and relationships between the terms that are introduced in this section.

Definition 9 (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 the 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 temporal correlation. If an event e precedes e' in the CFG, then in an enactment the firing of e *might* cause the firing of e'. Precedence, however, is not causality: the firing on an event does not necessarily cause the firing of its successors. For example, the event that represents the taking of the

```
1  chor [chor1] (
2    [chor1body] {
3      chor [chor2] (
4        [chor2body] {
5          parallel [prl1] do
6            [prl1branch1] {
7              [mex1] p2 → m1 to p1
8            }
9          and
10           [prl1branch2] {
11             [mex2] p3 → m2 to p1 , p2
12           };
13         [mex3] p3 → m3 to p1 , p2
14       }
15       | *:
16         skip [skip1]
17     );
18     chor [chor3] (
19       [chor3body] {
20         choice [choice1] p1
21         either
22           [choice1either] {
23             throw [throw1] e1
24           }
25         or
26           [choice1or] {
27             [mex4] p1 → m4 to p2 , p3
28           }
29       }
30     )
31   }
32   | e1 :
33     throw [throw2] e2
34   | *:
35     skip [skip2]
36 )
```

Figure 16: The running example.

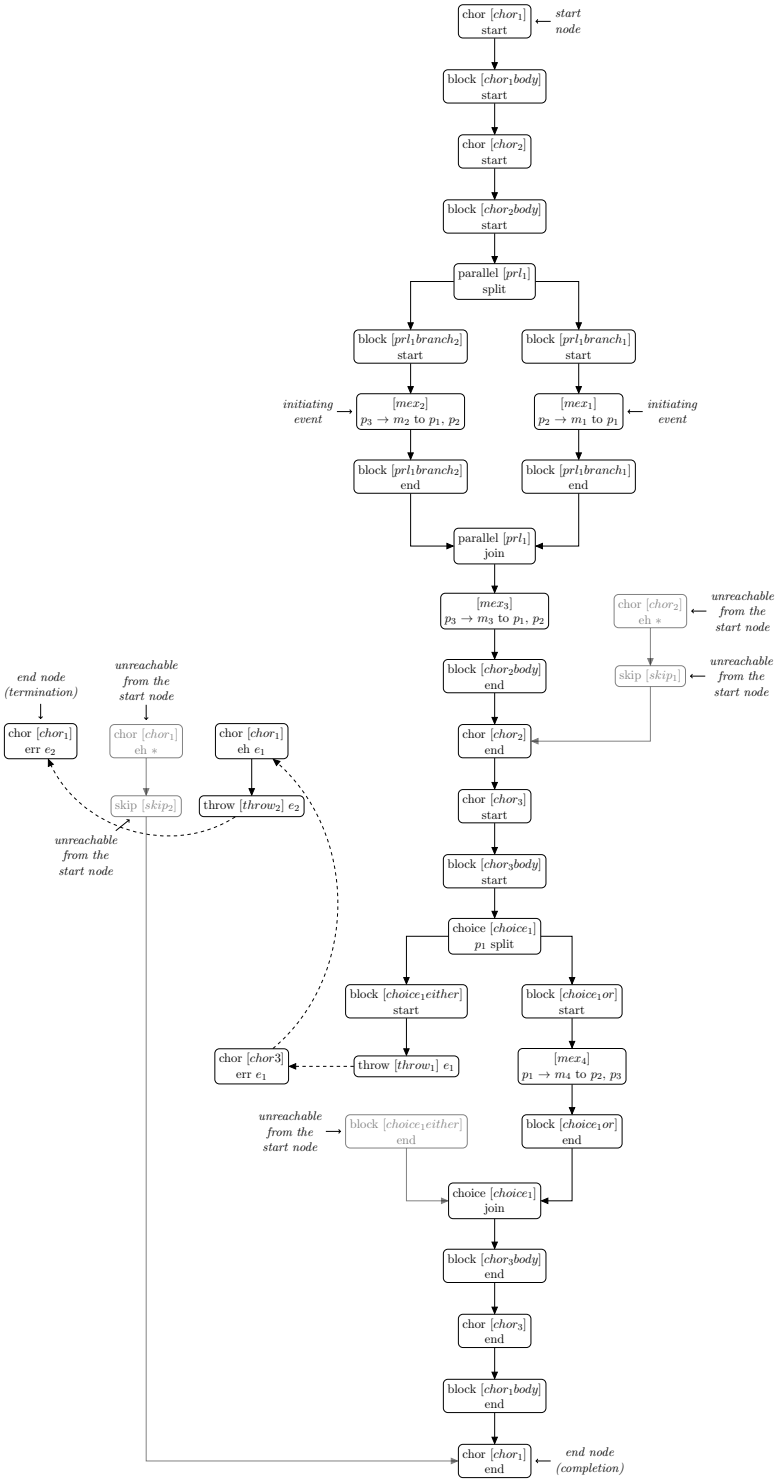


Figure 17: Control Flow Graph of the choreography shown in Figure 16.

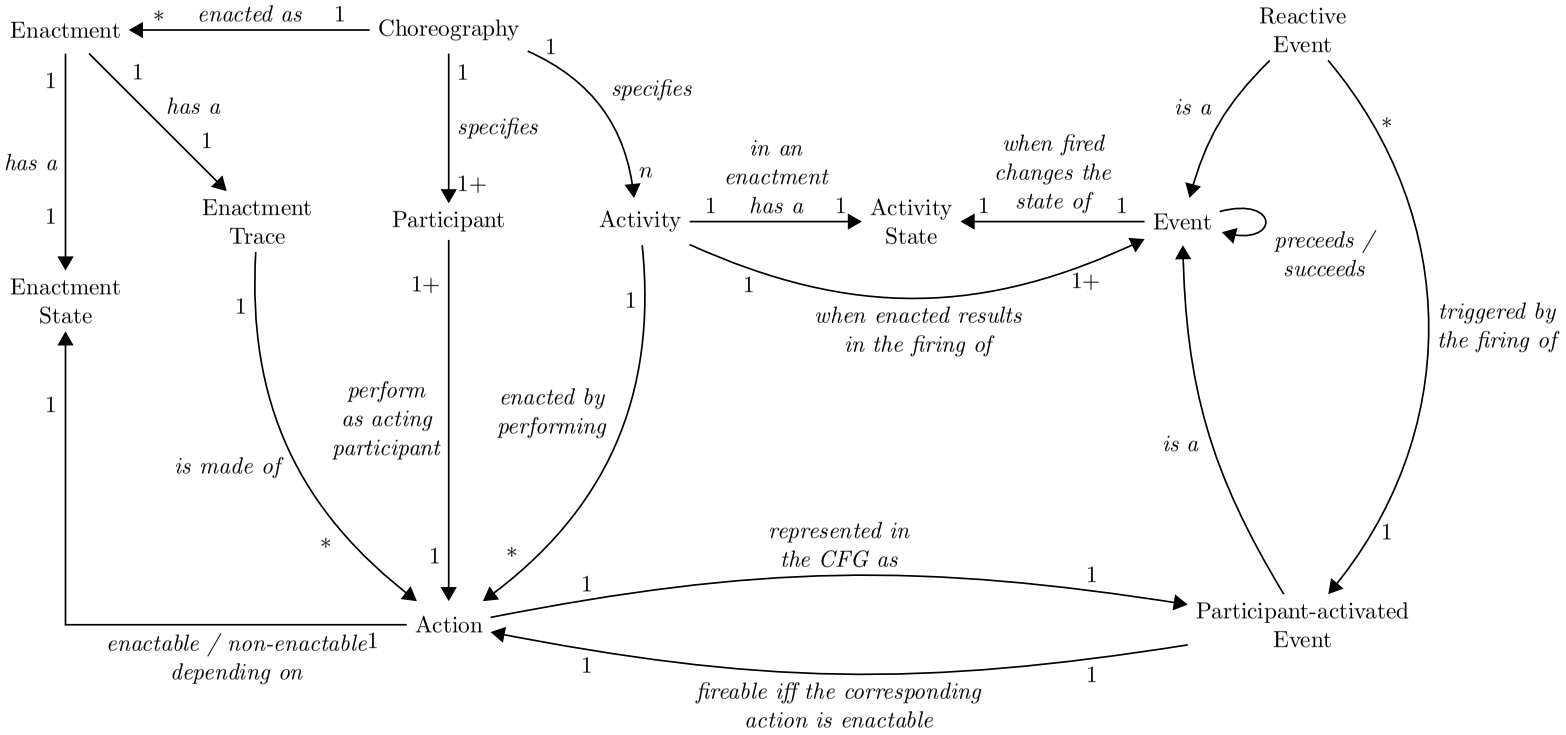


Figure 18: Terminology related to enactments, actions and events.

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 of those start events of the branches may occur. (Actually, it could even be the case that *none* will occur, if the choice activity is terminated because of the propagation of an exception thrown by a concurrently enacted activity.)

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

Definition 10 (Participant-Activated Events and Participant-Activated 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*.

Straightforwardly, each type of participant-activated events corresponds to one of the types of actions specified in Definition 1. Consistently with the fact that the reception of a message by a recipient is not considered an action (see Definition 1), it is not represented as an event either.

Definition 11 (Reactive Events and Reacting Nodes)**.** *Reactive events* are fired as a consequence of the firing of participant-activated events. The types of reactive events are all those not explicitly denoted as participant-activated in Definition 10. 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 19 obtained from a straightforward choreography.

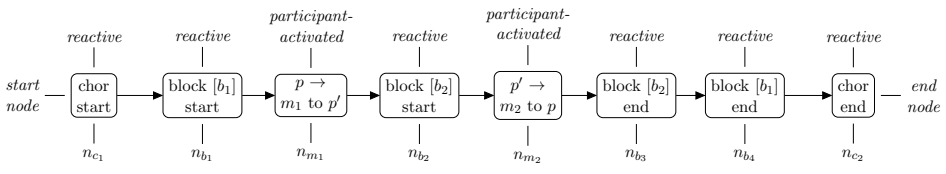


Figure 19: A CFG exemplifying reactive and participant-activated nodes.

The enactment of the choreography represented by the CFG in Figure 19 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 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.

Definition 12 (Choreography Violations)**.** An action 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 a been appended to the enactment trace.

For example, considering the CFG in Figure 19, 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, does not allow it. 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 13 (Enactable Actions)**.** An action a is *enactable* in the enactment state χ if and only if 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 14 (Fireable Events)**.** The participant-activated event e_{a} associated with the action a is *fireable* in the enactment state χ if a is enactable in χ.

5 Awareness Models of ChorTex Choreographies

This section discusses how to create an AWM by annotating the CFG of a ChorTex choreography (created as explained in Section 4.1) with information on the participant awareness. The current section is structured as follows. Section 5.1 discusses the concept of participant awareness, while Section 5.2 explains how the participant awareness is calculated.

5.1 Participant Awareness in Choreographies

As introduced in Section 2.2, a ChorTex choreography is fundamentally the intensional specification of sequences of actions such as message exchanges

and internal decisions, that the participants are tasked to perform during enactments. The CFG of a ChorTex choreography represents the ordering of the firing of events representing e.g. the beginning and completion of an activity, and are fired during an enactment as a result of the actions performed by the participants (see Section 4.2).

When enacting a choreography, the participants are subject to the phenomenon called *blindness* [DS08, DDC09], i.e. they might be able to observe only some of the actions that are performed. Because of the blindness, it might be the case that some participants have not enough information on how the enactment proceeds 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, or that performing a certain action violates the choreography. Put it in terms of events, a participant might be unable to observe sufficient events to play its role as specified, and this is symptomatic of the unrealizability of the choreography that is being enacted (see Section 3). 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 choreography, which is the basis for the verification of the strong realizability of a choreography presented in Section 6.

Definition 15 (Observable Actions)**.** The participant p can observe the performing of an action a (equivalently, a is *observable* by p), if and only if:

- p is an acting participant of a (see Definition 1), or
- 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 a.

In other words, a participant is able to observe only those actions it performs as their acting participant, such as the dispatching of a message, or that have observable consequences for the participant, i.e. the reception of a message addressed to it. As a result, the participants have *local perceptions* on the enactment traces.

Definition 16 (Local Perceptions)**.** The *local perception* of the participant p of an enactment trace σ is the trace obtained by purging σ of the actions not observable by p (see Definition 15).

$$\pi\ (p, (\!|\,\mathsf{a}\,|\!) \circ \sigma) := \begin{cases} (\!|\,\mathsf{a}\,|\!) \circ \pi\ (p, \sigma) & \text{if } (\mathsf{a} = [mex]p_s \xrightarrow{m} p_{r_1}, \ldots, p_{r_n}) \wedge \big(p = p_s \vee p \in \{p_{r_1}, \ldots, p_{r_n}\}\big) \\ (\!|\,\mathsf{a}\,|\!) \circ \pi\ (p, \sigma) & \text{if } \big(\mathsf{a} = [o]^{\text{▒▒}}(p_1, \ldots, p_n)\big) \wedge p \in \{p_1, \ldots, p_n\} \\ (\!|\,\mathsf{a}\,|\!) \circ \pi\ (p, \sigma) & \text{if } (\mathsf{a} = [c]p \overset{?}{\mapsto} \mathbf{x}) \\ (\!|\,\mathsf{a}\,|\!) \circ \pi\ (p, \sigma) & \text{if } (\mathsf{a} = [i]p \overset{\circlearrowleft}{\mapsto} \mathbf{x}) \\ \pi\ (p, \sigma) & \text{otherwise} \end{cases}$$

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

The above defined function $\pi\ (p, \sigma)$ specifies how to project the local perception of the participant p from the enactment trace σ.

It should be noted that the local perception of a participant on an enactment trace does not necessarily represent the order in which the participant has observed the actions. For example, in the running example shown in Figure 16, 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 perception, participants have a different knowledge on which actions have been performed up to that point in an enactment, and therefore which events have been fired so far. The knowledge of a participant with respect of the firing of an event is modeled by the concept of *participant awareness*. We distinguish between the participant awareness of a participant *before* and *after* the firing of an enacting event e:

Participant awareness of p before e represents the capability of p of observing that e becomes fireable (see Definition 14).

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

The differentiation of the participant awareness between before and after the firing of an event e allows to describe situations in which a participant p knows that e may be fired in the current enactment state, causing the transition to another enactment state, but the actual firing of e is not observable by p. This is extremely important for enacting the choreography without violations. For example, it may be the case that the action 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 perceptions to decide when to perform the actions that they *believe* are enactable. Therefore p may unknowingly violate the choreography by performing a in the enactment state χ' because it did know that the enactment state had transitioned from χ to χ' due to the firing of e.

Table 4 presents the four possible *participant awareness states*, i.e. the "extents" of participant awareness that a participant may have about the becoming fireable and the firing of an event e.

Immediate awareness, denoted by "ia," means that the participant knows *as soon as it happens* that the event becomes fireable or is fired (awareness before and after the event, respectively). 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 awareness. The participant does not necessarily know that the event has become fireable

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

Table 4: The possible participant awareness states of the participant p with respect to an event e.

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 Temporal Logics (LTL), see e.g. [BK08]. For example, due to the asynchronous nature of message exchanges in 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.

An *unaware*, denoted by "ua," participant cannot observe at all an event becoming fireable or being fired. For example, a participant p that is neither sender nor recipient of a message m is unaware of participant-activated event that represents the dispatching of that message. Of course, due to some later interaction with the other participants, p might be able to "imply" that m has been dispatched. This is the case, for example, when the dispatching of m must have necessarily been occurred before another message m' is dispatched, and p can observe the dispatching of m'. However, this type of implication is not of interest to the realizability analysis proposed in this work, and 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 has not yet taken part in the enactment. (Notice that the participant might later become involved in the enactment, in which case the participant will be immediately-, eventually- or unaware of some other events and might *imply* that the events he is not-involved of have been fired; however, as explained before, this type of implication is not relevant to the ends of participant awareness, which is based on *observation*.) Non-involvement implies unawareness. In fact, 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.

5.2 Annotating Participant Awareness States

This section shows how to create AWMs by annotating CFGs generated from ChorTex choreographies (see Section 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 are exactly the same of the original CFG.

The annotations on the participant awareness sates associated with the nodes in the AWMs are called *node awareness states*.

Definition 17 (Node Awareness States)**.** In the AWM of a choreography specifying the participants $p_1, \ldots, p_m$, the *node awareness state* of a node n that represents the firing of the event e is defined as the following couple:

$$\Big(\mathrm{AW}_{\mathrm{fireable}}(n), \mathrm{AW}_{\mathrm{fired}}(n)\Big)$$

$\mathrm{AW}_{\mathrm{fireable}}(n)$ and $\mathrm{AW}_{\mathrm{fired}}(n)$ are sets comprising m items, each of them representing the participant awareness state of one participant with respect to the

event e becoming fireable and e being fired, respectively. The $\mathrm{AW}_{\mathrm{fireable}}(n)$ and $\mathrm{AW}_{\mathrm{fired}}(n)$ of an arbitrary node n are both formally defined as follows:

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

That is, they contain *exactly one* participant awareness state associated to each of the participants specified by the choreography.

$\mathrm{AW}_{\mathrm{fireable}}(n, p)$ and $\mathrm{AW}_{\mathrm{fired}}(n, p)$ denote the participant awareness state of the participant p in $\mathrm{AW}_{\mathrm{fireable}}(n)$ and $\mathrm{AW}_{\mathrm{fired}}(n)$, respectively.

5.2.1 The Awareness Annotation Algorithm

Figure 20 presents the pseudo-code of the Awareness Annotation Algorithm (AAA), which calculates the node awareness states associated with the nodes of AWMs. The AAA has the classic structure of fixed-point algorithms on CFGs, and it begins with an initialization phase that assigns for each node and to all participants an initial "non-involved" $\mathrm{AW}_{\mathrm{fireable}}$ value. The function $\mathbb{P}(chor)$, defined in Figure 21, returns the set of participant identifiers appearing in the choreography *chor* by traversing its parsing tree. For each participant p, the $\mathrm{AW}_{\mathrm{fired}}$ of a node n is calculated by the function $f(n, p)$, which is discussed in detail in Section 5.2.3. 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 modifications of the node awareness states. At each iteration step, a node n is chosen at random to be processed. (Although, implementations of the AAA may employ heuristics e.g. based on the cycles in the CFGs; however, such heuristics fall outside the scope of this work, as they add nothing to the concepts here presented.) The processing of n consists of the following two steps, which are repeated for each participant p specified in the choreography. First, the $\mathrm{AW}_{\mathrm{fireable}}$ value of n is recalculated as the *safe approximation* of all the $\mathrm{AW}_{\mathrm{fired}}$ values of n's predecessors, which is calculated using the function $\bigcap\!\!\!\!\!\!\bigcap$ described in Section 5.2.2. (For reasons of understandability, we discuss how the safe approximation is calculated in Section 5.2.2.) Secondly, $\mathrm{AW}_{\mathrm{fired}}(n, p)$ is updated with the outcome of $f(n, p)$, which in some cases is calculated on the basis of the updated value of $\mathrm{AW}_{\mathrm{fireable}}(n, p)$ that results from the previous step.

Figure 22 presents the AWM resulting from the application of the AAA to the CFG of the running example shown in Figure 16. The node awareness states are represented as labels attached to the respective nodes. Notice that there are no $\mathrm{AW}_{\mathrm{fireable}}$ and $\mathrm{AW}_{\mathrm{fired}}$ values annotated on the nodes that are unreachable from the start node, which are drawn less markedly in Figure 16. The nodes unreachable from the start one are ignored when calculating

the node awareness states because they represent events that never become fireable and are never fired in any enactment of the choreography. In the pseudo-code of the AAA, this is realized by the "reachable from *start*" clauses on Line 9 and Line 20.

Awareness Annotation Algorithm

```
1  function generateAwarenessModel (choreography chor)
2      returns awareness model {
3    /* generate chor's CFG as described in Section 4.1 */
4    awareness model awm := generateCFG(chor);
5    /* retrieve the start node of the AWM */
6    node start := startNode(awm);
7
8    /* initialization */
9    foreach node n in ℕ(awm) reachable from start do
10     /* ℙ(chor) returns the set of participants */
11     /* specified by the choreography chor (see Figure 21) */
12     foreach p in ℙ(chor) do
13       AW_fireable(n, p) = ni;
14       AW_fired(n, p) = f(n, p);
15     end foreach
16   end foreach
17
18   /* iteration until a fixed-point is reached */
19   repeat
20     foreach node n in ℕ(awm) reachable from start do
21       /* update AW_fireable as safe-approximation of */
22       /* the AW_fired of the successors of n */
23       foreach p in ℙ(chor) do
24         /* of the predecessors of n (see Section 5.2.2) */
25         AW_fireable(n, p) = ⨅_{n' ∈ pred(n)} (AW_fired(n', p));
26         /* update output (see Section 5.2.3) */
27         AW_fired(n, p) = f(n, p);
28       end foreach
29     end foreach
30   until no AW_fired(n, p) changes
31
32   return awm;
33 }
```

Figure 20: Pseudo-code of the Awareness Annotation Algorithm.

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

Figure 21: Recursive definition of the function $\mathbb{P}(\mathbf{A})$ for extracting the participants involved in an activity $\mathbf{A}$.

5.2.2 Safe Approximation of $\mathrm{AW}_{\mathbf{fireable}}$ Values of Nodes

In the iteration phase of AAA, the $\mathrm{AW}_{\mathrm{fireable}}$ value of a node is recalculated as the safe approximation of the $\mathrm{AW}_{\mathrm{fired}}$ values of its predecessors of n. The safe-approximation of $\mathrm{AW}_{\mathrm{fired}}$ of a set of nodes is calculated by the function $\bigoplus$ (read "meet") defined Figure 23.[9] The safe approximation is calculated on the basis of the following principles:

1. Immediate awareness implies eventual awareness (see Section 5.1);
2. Non involvement implies unawareness (see Section 5.1);
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 $\bigoplus$, i.e. 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 4.2, an event becomes fireable when one or more of its predecessors have been fired (see Definition 14). If the participant p is unaware of the firing any of the predecessors of the event e, then there are some enactments of the choreography in which p is unaware of e becoming fireable. Therefore, the safe approximation is marking p as unaware in $\mathrm{AW}_{\mathrm{fireable}}(n)$.

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

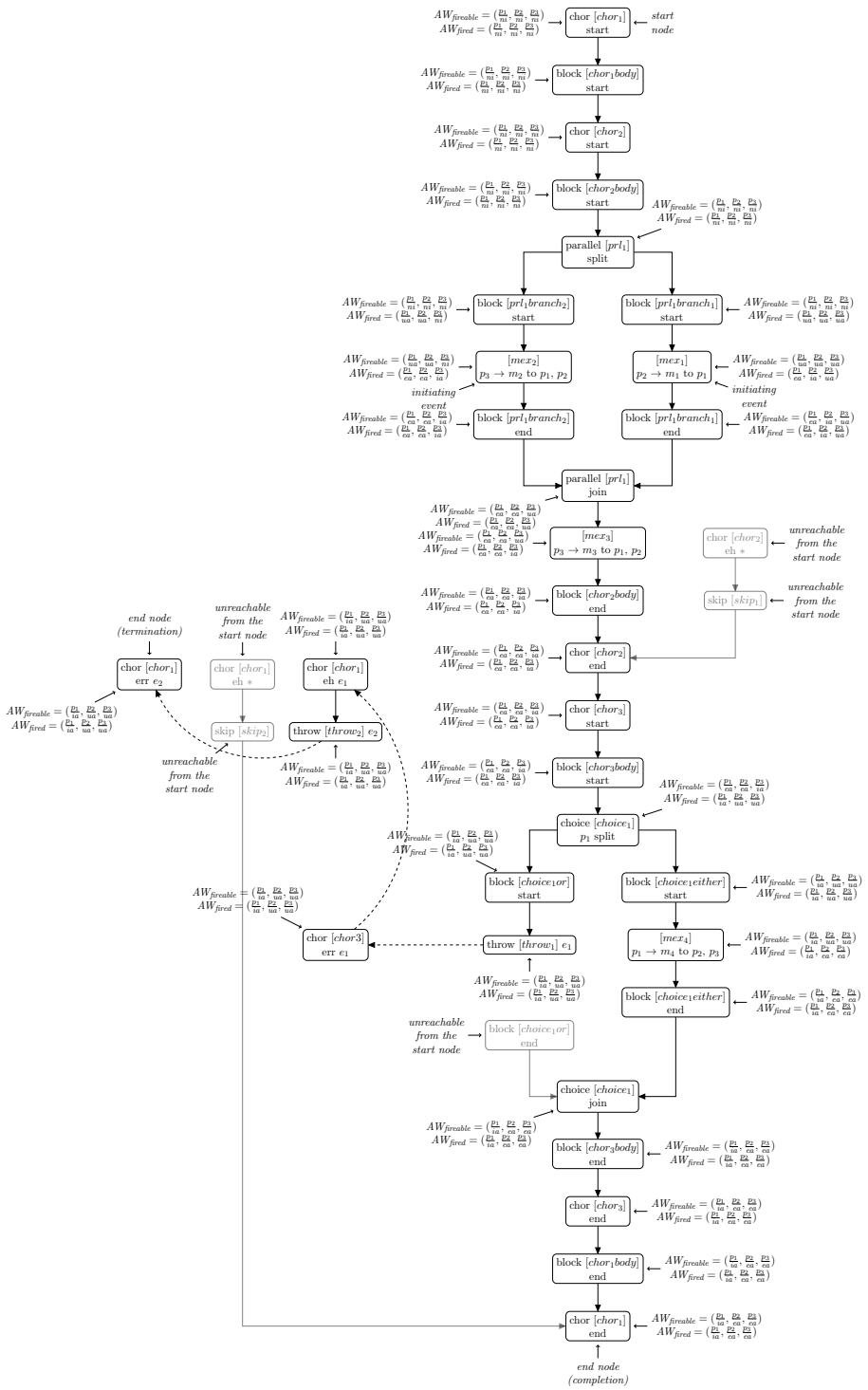


Figure 22: Awareness Model obtained by applying the Awareness Annotation Algorithm to the Control Flow Graph shown in Figure 17.

5.2.3 Updating the $\mathrm{AW}_{\mathbf{fired}}$ Value of Nodes

The value of $\mathrm{AW}_{\mathrm{fired}}(n, p)$ of a node n for the participant p is calculated with the function $f(n, p)$. The function $f(n, p)$ is defined in Table 5 case-based on the different types of AWM nodes. In practice, 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 AAA on each node n once for each of the participants in the choreography. In some cases, $f(n, p)$ simply returns the value of $\mathrm{AW}_{\mathrm{fireable}}(n, p)$, meaning that the traversal of nodes of those types 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, and the start, end, exception handling and error propagation nodes of choreographies.

After the traversing of a node representing the dispatching of a message exchanges, the sender is immediately aware. The recipients, instead, are eventually aware because the messages are processed asynchronously. The participants that were not involved before traversing the node remain not-involved after it. Otherwise, the participants are unaware.

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 2.1). Participants that were not involved before the opaque activity stay not involved. Otherwise, the participants that were involved before the opaque activity, but that do not take part in it, become unaware.

The nodes representing local decisions for iteration and choice activities modify the participant awareness in the same way. Namely, since the decision is local, the decision maker is immediately aware of it. All participants that

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

$$\bigoplus(\alpha_1, \ldots, \alpha_m) := \begin{cases} \mathrm{ia} & \text{if } \forall\, j \in [1, m] : \alpha_j = \mathrm{ia} \\ \mathrm{ea} & \text{if } \big(\exists\, j \in [1, m] : \alpha_j = \mathrm{ea}\big) \wedge \\ & \quad \big(\forall\, i \in [1, m] : \alpha_j = \mathrm{ia} \vee \alpha_j = \mathrm{ea}\big) \\ \mathrm{ni} & \text{if } \forall\, j \in [1, m] : \alpha_j = \mathrm{ni} \\ \mathrm{ua} & \text{otherwise} \end{cases}$$

Figure 23: Definition of the $\bigoplus$ function that calculates for a participant p the safe-approximation of the $\mathrm{AW}_{\mathrm{fired}}$ values of multiple nodes.

AWM node n	**Definition of** $f(n,p)$
$p_s \to m$ to $p_{r_1}, \ldots, p_{r_n}$	$\begin{cases} \text{ia} & \text{if } p = p_s \\ \text{ea} & \text{if } p \in \{p_{r_1}, \ldots, p_{r_n}\} \\ \text{ni} & \text{if } \left(p \notin \{p_s\} \cup \{p_{r_1}, \ldots, p_{r_n}\}\right) \wedge \text{AW}_{\text{fireable}}(n,p) = \text{ni} \\ \text{ua} & \text{otherwise} \end{cases}$
opaque $(p_1, \ldots, p_m)$	$\begin{cases} \text{ea} & \text{if } p \in \{p_1, \ldots, p_m\} \\ \text{ni} & \text{if } p \notin \{p_1, \ldots, p_m\} \wedge \text{AW}_{\text{fireable}}(n,p) = \text{ni} \\ \text{ua} & \text{otherwise} \end{cases}$
block start	$\begin{cases} \text{ua} & \text{if block is branch of parallel activity } \mathbf{A} \wedge \text{AW}_{\text{fireable}}(n,p) = \text{ni} \wedge \\ & \quad p \text{ is acting participant in another branch of } \mathbf{A} \\ \text{AW}_{\text{fireable}}(n,p) & \text{otherwise} \end{cases}$
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 5: Thedefinition of $f(n,p)$ for the node n and the participant p, case-based on the various types of AWM nodes.

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. If the block is not a branch of a parallel activity, then no participant awareness states are modified. Otherwise, the block is a branch of a parallel activity, and we must take extra care with participants that are currently not involved. 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 a parallel activity, but it becomes involved in one of its branches, the flow algorithm would "copy" the information about p's non involvement on all its 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. In the definition of $f\,(n, p)$ shown in Table 5, 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 non involved in the $\mathrm{AW}_{\mathrm{fireable}}\,(n)$, and p is an acting participant any other branch of the parallel activity. This is the case, for example, of the AWM node `block [`$prl_1branch_2$`] start` in the running example shown in Figure 22. In fact, the participants p_1 and p_2 are unaware in $\mathrm{AW}_{\mathrm{fired}}\,($`block [`$prl_1branch_1$`] start`$)$ – instead of non-involved as reported in $\mathrm{AW}_{\mathrm{fireable}}\,($`block [`$prl_1branch_2$`] start`$)$ – because p_1 and p_2 are acting participants in the other branch of the parallel activity prl_1, i.e. the block activity $prl_1branch_1$.

5.2.4 Computational Complexity of the AAA

The AAA is a "classic" iterative fixed-point algorithm. Similar fixed-point flow algorithms, e.g. the ones for calculating dominance and post-dominance on CFGs, perform a number of iterations to update the values associated to the nodes proven to be between $\mathcal{O}(|\mathbb{E}\,(awm)| \times \log|\mathbb{E}\,(awm)|)$ (see [Tar74]) and $\mathcal{O}(|\mathbb{N}\,(awm)|^2)$ (see e.g. [CHK01]), where $\mathbb{E}\,(cfg)$ and $\mathbb{N}\,(awm)$ denote the sets of control-flow edges and nodes of the CFG cfg, 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 all the couples of nodes, which is known to have upper-bound $\log^2|\mathbb{N}\,(awm)|$.

Each invocation of the merge function $\pitchfork$ has complexity linear to (1) the number participants and (2) 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 4.1), so we can approximate the upper bound complexity of of $\pitchfork$ to $\mathcal{O}\,(|\mathbb{P}\,(chor)|)$, where $|\mathbb{P}\,(chor)|$ denotes the cardinality of the set of participants, i.e. 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 when 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 $\mathcal{O}\,(|\mathbb{A}\,(chor) - 4|)$. The reason is simple: the activities that can specified in the other branches of the parallel can be at most $|\mathbb{A}\,(chor) - 4|$, because for sure they do not contain the current branch, the activity that it necessarily contains (see ChorTex's extended BNF in Section 2.1), the parallel activity in which the current branch is nested, and the root choreography.

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

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

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

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

An over-approximation of 4 on the count of activities and a complexity of $\log^2 |\mathbb{N}\,(awm)|$ do not matter, 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.") Finally, we can observe that the number of nodes in a CFG is roughly linear with the number of activities specified by the choreography. (It is not possibly to quantify precisely the ratio between activities and CFG 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). Therefore, the estimation of the upper-bound complexity of the AAA can be further conservatively simplified to:

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

6 Verifying the Strong Realizability of ChorTex Choreographies through Awareness Constraints

This section introduces the constraints on participant awareness states of an AWM of a ChorTex choreography that, if verified, ensure the strong realizability of the latter. We assume the *goodwill* of the participants, i.e. they do not knowingly violate the choreography.[10] In other words, if a participant is aware that performing an action in the current enactment state would violate the choreography, it will not perform that action. Under the assumption of the participants' goodwill, the strong realizability of a ChorTex choreography is verified if the three following *strong-realizability requirements* are met by its participants:

Know when to act: The acting participants of every action are immediately- or eventually aware of the latter becoming enactable;

Know when not to act: The acting participants of every action are immediately aware of the latter becoming not enactable;

Know when the role is over: In any enactment, each participant eventually knows when its part in it is over, i.e. if no matter how the enactment proceeds, no further actions are required by the participant and no further messages will be received.[11]

The first two of the requirements above are of straightforward explanation. Firstly, if a participant with goodwill knows when it can perform its actions and when not, 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 hence they will eventually perform it.

The third requirement, i.e. that the participants when their roles are over, has a less immediate rationale and comes for the language-equivalence in terms of conversations between the composition of the roles and the choreography that is required by the definition of strong realizability. The reason is the following. As shown for example in [KP06], a choreography can be represented as a State-Transition System (STS). The roles specified by the choreography can also be modeled as separate STSs. The composition of the roles, therefore, is a composite STS the state space of which is the Cartesian product of the state spaces of the various STSs of the roles, and removing those that are not reachable to the transitions in the single STSs. The final states of the composite STS are those in which *all* the separate roles are

[10] At the best of our knowledge, virtually every work in the state of the art on realizability, with the exception of [BTZ12], builds on this same assumption.

[11] The authors have pondered at length whether to name this requirement "He's dead, Jim." While not been a particularly academic name, it does sound eerily appropriate.

themselves in a final state. Recall that the behavioral equivalence that is adopted in Definition 8 is language-equivalence. Put it in terms of STSs, language-equivalence requires (among other things) that all and only the conversations that lead the STS of the choreography to a final state also lead the composite STS of the roles to a final state. Naturally, a role that has not begun is in a final state because the corresponding participant has not yet been involved in the enactment. Therefore, in order for a choreography to be strongly realizable, in every possible enactment the participants that have been involved in it must know when their roles are completed.

In order to verify the strong realizability of a ChorTex choreography, the three strong-realizability requirements are "translated" to *awareness constraints*.

Definition 18 (Awareness Constraints)**.** An *awareness constraint* is a first-order logic predicate evaluated on the node awareness states of an AWM. Besides the usual first-order logic operators, an awareness constraints admits the following predicates which test the participant awareness state of a participant p with respect to the becoming fireable and the firing of the event represented by the node n of the AWM:

$$\mathrm{AW}_{\text{fireable}}(n, p) = \alpha \qquad \text{(Participant-awareness state in } \mathrm{AW}_{\text{fireable}}\text{)}$$

$$\mathrm{AW}_{\text{fired}}(n, p) = \alpha \qquad \text{(Participant-awareness state in } \mathrm{AW}_{\text{fired}}\text{)}$$

$$\mathrm{SC}_{\text{out}}(n, p) = \textsc{can} \qquad (p \text{ "spots" the end of its role from the node } n)$$

(The latter predicate is reported here for completeness; the explanation of its meaning requires a large amount of groundwork, and is postponed to Section 6.3.) As in Section 5.2, the symbol α denotes one of the four participant-awareness states described in Section 5.1, i.e.:

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

Each awareness constraints is *attached* to one node of the AWM for allowing "short-hand" notations. In particular, in an awareness constraint attached to the AWM node n, the following couples of predicates are equivalent:

Long-hand	**Short-hand**
$\mathrm{AW}_{\text{fireable}}(n, p) = \alpha$	$\mathrm{AW}_{\text{fireable}}(p) = \alpha$
$\mathrm{AW}_{\text{fired}}(n, p) = \alpha$	$\mathrm{AW}_{\text{fired}}(p) = \alpha$
$\mathrm{SC}_{\text{out}}(n, p) = \textsc{can}$	$\mathrm{SC}_{\text{out}}(p) = \textsc{can}$

That is, the short-hand version of the predicates "implies" the node whose node awareness state is tested to be the one to which the awareness constraint is attached.

Figure 24 exemplifies on the running example the types of awareness constraints that are presented in Section 6.1 through Section 6.3.

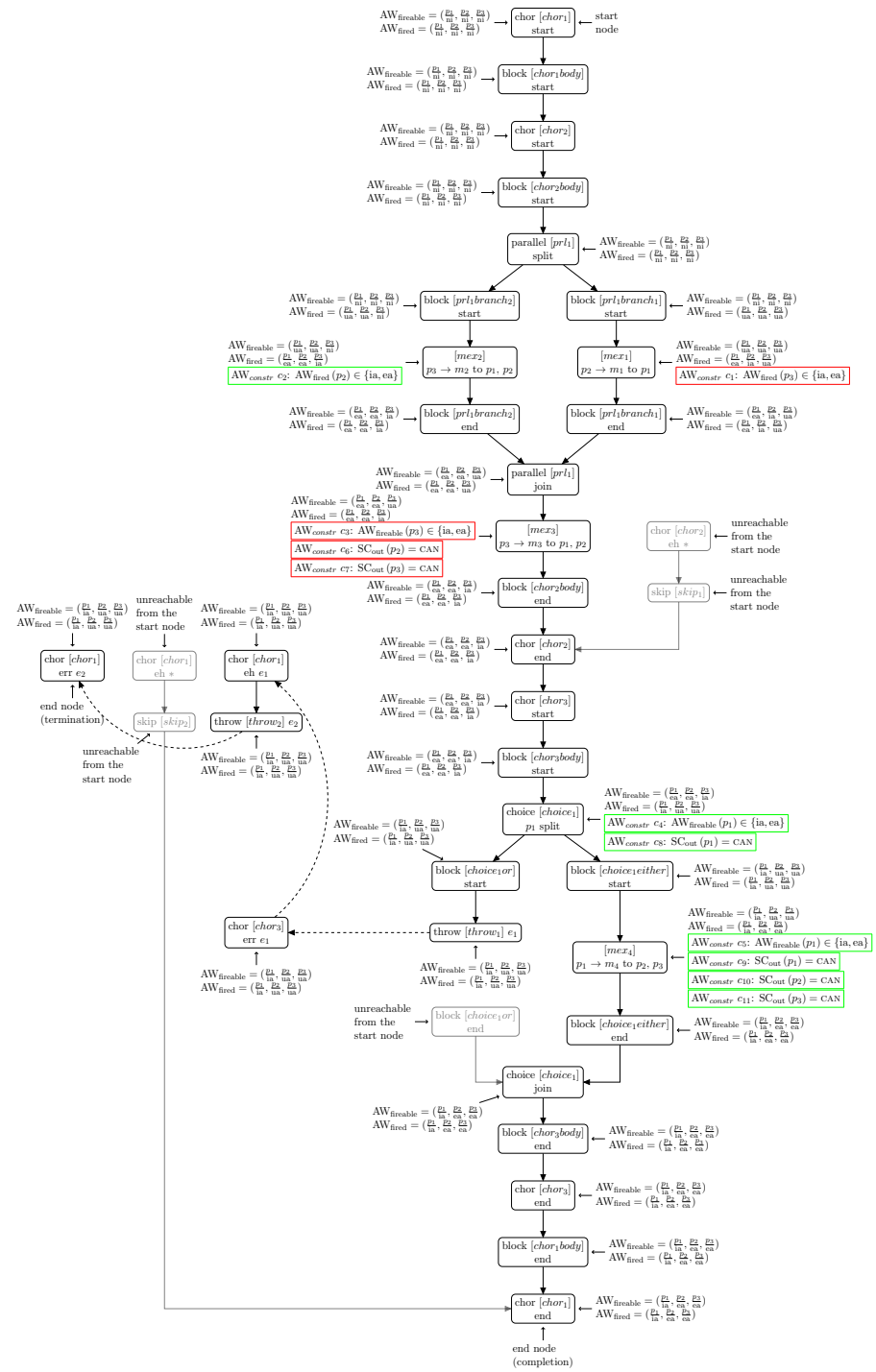


Figure 24: The Awareness Model presented in Figure 16, augmented with the awareness constraints; the satisfied awareness constraints are framed in green, the unsatisfied ones in red.

6.1 Know When to Act

The "know when to act" principle is translated to awareness constraints attached to nodes of the AWM that represent participant-activated events. The are two types of such awareness constraints, depending on whether the action associated with the participant-activated event is initiating or not.

6.1.1 Non-initiating actions

Let n be an AWM node representing the firing of a participant-activated event e_{a} associated with the non-initiating action a. To verify that the acting participants of a "know when to act," n has attached the following awareness constraint requiring each acting participant of a to be either immediately or eventually aware of a becoming enactable (see Definition 13):

$$\forall\, p \in \mathit{actingParticipants}\,(\mathsf{a}) : \mathrm{AW}_{\mathrm{fireable}}\,(p) \in \{\mathrm{ia}, \mathrm{ea}\}$$

The function $\mathit{actingParticipants}\,(\mathsf{a})$ returns the acting participants of the action a (see Definition 1). The rationale of this awareness constraint is the following. The acting participants must be aware of when they can perform the action, i.e. when it becomes enactable (hence the constraint on $\mathrm{AW}_{\mathrm{fireable}}$). If the acting participants are immediately aware of when the action become enactable, they will be able to perform the action as soon as it would not violate the choreography. If they are eventually aware, they will be able to perform it at some point in the future. If the acting participants are unaware or not-involved, they can never perform that action for the risk of violating the choreography, and this constitutes a realizability defect.

An example of this type of awareness constraint is c_3 in Figure 24. The participant p_3 is unaware of the fact that it can dispatch of m_3, and therefore it never will, thus deadlocking the enactment. On the contrary, the decision maker p_1 of the choice activity choice_1 is eventually aware of the fact that it is required of deciding which branch to enact, and assuming that it will perform the decision in a finite amount of time, this guarantees the progress of the enactments (see awareness constraint c_4 in Figure 24) .

6.1.2 Initiating actions

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, are not possible due to the constructs of 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. As a consequence, 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 participants "know when to act." 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, and to perform it is reasonable not to require prior awareness, i.e. knowledge 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 to determine how the acting participants of the initiating action decide to perform it.

On the contrary, in the case of multiple, non-exclusive initiating actions, awareness constraints are required to ensure that all the initiating actions can be performed *irrespective to which one is the first to occur in an enactment*. In the running example (see Figure 24) 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 firstly dispatched, and then m_2, or vice-versa. For both the senders of the two message exchanges, namely p_2 and p_3, to be able to dispatch their message with the certainty of not violating the choreography, it must be the case that the dispatching of one of the messages "provides enough awareness" to the sender of the other to be able to dispatch the other message. Unfortunately, this is not the case of 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 from the example reported above, 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. Therefore, enactments can be begun by any of the initiating actions without resulting in deadlocks, exactly as required by the choreography. Put formally, given the set of initiating actions $\mathfrak{A} := \mathsf{a}_1, \ldots, \mathsf{a}_m$ of the choreography, each of the

nodes $n_{\mathsf{a}_1}, \ldots, n_{\mathsf{a}_m}$ representing the participant-activated events associated with the initiating actions $\mathsf{a}_1, \ldots, \mathsf{a}_m$ has associated the following awareness constraint:

$$\forall\, p \in \Big(\bigcup_{\mathsf{a}' \in \mathfrak{A} \setminus \{\mathsf{a}\}} \mathit{actingParticipants}\,(\mathsf{a}') \Big) : \mathrm{AW}_{\mathrm{fired}}\,(p) \in \{\mathrm{ia}, \mathrm{ea}\}$$

The awareness constraint above can be paraphrased in natural language as follows: the acting participants of every initiating action other than a must be eventually- or immediately aware of the performing of a. Thus, 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.

6.2 Know When Not to Act

When required to perform an action, i.e. generating a message or taking a decision, a participant is not required to take that action *immediately*. In the life-cycles of ChorTex activities presented in Section 2.2.2, this is represented by the PENDING states of message-exchange (Figure 6), choice (Figure 10) and iteration (Figure 11) activities. That is, some time may pass between the moment an action becomes enactable (see Definition 13), and when it is actually enacted. The awareness constraints that realize the "know when to act"-requirement (presented in Section 6.1) 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 2). The operational semantics of exception throwing specifies that, as soon as an exception is thrown, all the other activities that are being enacted as branches of a parallel activity are terminated. In such situations, the unawareness of acting participants with respect to the current non-enactability of their actions may mislead them into violating the choreography.

Consider the choreography shown in Figure 25. After that the participant p_1 has dispatched a message of type m_1 to p_2, 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. Currently, there are two enactable actions: the dispatching of a message of type m_2 by p_2 to p_1, and the decision by p_1 on which branch of the choice activity c_1 to enact. In an enactment in which p_1 decides "true" in the choice activity c_1 before p_2 dispatches the message m_2, the latter action becomes non-enactable. In this case, if the sender of mex_2, i.e. p_2, dispatches m_2 to p_1 despite the fact that the exception e_1 has been thrown, the choreography is violated.

```
1  chor [chor1] (
2      {
3          [mex1] p1 → m1 to p2;
4          parallel [prl1] 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 25: A choreography in which actions can change from enactable to non-enactable due to the throwing of exceptions.

6.2.1 Identifying Concurrently-Acting Participants

To verify that a choreography does not have realizability defects such as those of the choreography in Figure 25, we need awareness constraints that verify that the participants that may act in other concurrently-enacted branches are *immediately aware* of the throwing of exceptions or their propagation from nested choreographies. Figure 26 presents an algorithm that derives 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 *parent* (**A**) returns the activity in which **A** is nested. If **A** is the root choreography (which, by definition, has no parent activity), *parent* (**A**) returns NULL. The function *sourceActivity* (n), instead, returns the activity from which the CFGs node n was generated (see Section 4.1).

The algorithm in Figure 26 is divided in two phases. First, it identifies the parallel activity that encompasses all those activities that may be terminated when the *propagates exception* node n is traversed (i.e. when the event represented by n in the AWM is fired). This parallel activity, called *outer parallel*, is the last one encountered navigating upwards – i.e. from nested activities to those encompassing them – 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

```
1  /* n is a propagates exception node */
2  /* the algorithm attaches the newly generated constraint to n */
3  function deriveAwarenessConstraint (node n)
4          returns constraint {
5      /* A is the activity from which the propagates exception node */
6      /* n has been generated (i.e. a throw activity or a choreography) */
7      activity A := sourceActivity(n);
8
9      /* identification of the "outer" parallel activity */
10     parallel activity outerParallel := NULL;
11     activity A' := parent(A);
12     /* A' is NULL if A is the root choreography */
13     /* the iteration terminates when A' is a choreography */
14     /* (not necessarily the root one) */
15     while (A' not NULL && A' is not choreography) do
16         if (A' is parallel do A_1 and ... and A_n) then
17             outerParallel := A';
18         end if
19         A' := parent(A');
20     end while
21
22     /* identification of the nodes that represent actions that */
23     /* may be made non-enactable after the traversing of */
24     /* the propagates exception node n */
25     if outerParallel not NULL then
26         /* the set P contains the acting participants of actions */
27         /* that can be concurrently enacted */
28         set P := ∅;
29         /* retrieve/generate chor's CFG sub-graph */
30         /* as described in Section 4.1 */
31         foreach node n' in generateCFG(outerParallel) do
32             if (not n' dominates n &&
33                     not n dominates n') then
34                 P := P ∪ actingParticipants(n');
35             end if
36         end foreach
37         return new constraint c := ⋀_{p ∈ P} AW_fired(p) = ia;
38     end if
39 }
```

Figure 26: Pseudo-code for deriving the awareness constraint attached to the *propagates exception* node n.

that no outer parallel is found, i.e. if:

- 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);
- 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 above cases, no awareness constraint is needed for n. The reason is that, since no outer parallel activity is found, there are no concurrently-enacted actions that might be rendered non-enactable by the propagation of the exception.

In case an outer parallel is found, the second part of the algorithm identifies which actions might be made non-enactable by the traversing of the *propagates exception* node n. These actions are all those represented by some node in the CFG sub-graph of the parallel activity identified before, with the exception of the nodes dominated n (because they either have already been enacted before n is traversed) or those that dominate 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 constraint is composed by requiring that each acting participant of at least one of those actions is immediately-aware of the firing of the event represented by n.

6.2.2 Computational Complexity

The algorithm in Figure 26 requires the traversing of the parsing tree of the choreography, and it 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 which, once again, we 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 to the number of nodes in the CFG, and therefore this upper-bound can be over-approximated to:

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

More precise approximations are possible but not worth the effort, because we are considering the computational upper-bound, and this is not the highest among the various algorithms that compose the realizability analysis (see Section 7).

6.3 Know When the Role is Over

A final state is an enactment state (see Definition 5) in which no further action is *required* of the participants. Notice that a deadlock is not a final state; a deadlock, in fact, is an enactment state in which further action *is* required of the participants, but they *cannot perform it*. The reaching of a final state in the enactment of a choreography is represented in the respective AWM by the traversing of an end node. Choreographies may admit multiple final states, each representing a different *outcome*. As a consequence, AWMs may have multiple end nodes, each associated with a different final state. For example, the AWM shown in Figure 24 has two end nodes. The node `chor` [$chor_1$] `end` represents the successful completion of the choreography. Instead, `chor` [$chor_1$] `err` e_2 represents the termination of the choreography because of an exception of type e_2 that cannot be handled and that propagates outside the root choreography.

If verified on a choreography, the awareness constraints presented in Section 6.1 and Section 6.2 guarantee that exist participant implementations the composition of which is trace-equivalent to the choreography, i.e. it can enact *exactly* (i.e. all and only) the conversations specified by the choreography. However, the definition of strong realizability for ChorTex choreographies requires *more* than trace-equivalence: it requires language-equivalence (see Definition 8). Language-equivalence is a stronger relation than trace-equivalence because it additionally requires that every conversation that leads the choreography to a final state also leads the composition of participant implementations to a final state, and vice-versa [BIM95]. This additional requirement is the reason for the the "know when the role is over"-requirement, which is verified by the awareness constraints presented in the reminder of this section.

Adopting the formalization proposed in [KP06] of choreographies, participant implementations and their compositions as STSs, a composition of participant implementations reaches a final state if and only if each of the composed participant implementation reaches a final state. The final states of a participant implementation are its internal states that represent the completion of the role in some enactments. Naturally, participants that are not yet involved in an enactment before the latter ends need not know that their role in that enactment is over, since, in fact, it had not even yet begun.

Unlike the case of choreographies, the reaching of a final state by the participants is not explicitly represented in the AWMs. This is further complicated by the fact that, when the enactment is completed, all roles are over, but not all the participants are necessarily aware of it. Consider the choreography shown in Figure 27. An enactment is completed when p_1 decides not to iterate the activity *loop* further. Since p_1 takes the decision that immediately results in the completion of the enactment, it knows when the enactment is over, and thus that also its role is. On the other hand, p_2

```
1 chor [chor1] ({
2     iteration [loop] p1 do {
3         [mex1] p1 → m1 to p2
4     }
5 })
```

Figure 27: A choreography in which p_2 does not know when its role is over.

does not know when the enactment is over: its role is to receive an unspecified number of messages of type m_1, as long as they they keep being sent. When p_1 decides not to further iterate the activity *loop*, p_2 will still wait indefinitely for more messages that will never be dispatched. This "hanging" of p_2 violates the "know when the role is over"-requirement for strong realizability, and thus constitutes a realizability defect: since p_2's participant implementation cannot reach a final state by the time the enactment ends, the composition of the participant implementations of p_1 and p_2 does neither, and therefore the composition of the participant implementations and the choreography are not language-equivalent.

To avoid realizability defects like the one shown in the previous example, we need to verify that, by the time an enactment ends, all the participants involved in it know that their roles are over. A participant knows that its role in an enactment is over by implying that its role is over based on the events it can observe (see Definition 15) and on how the choreography is specified. The explanation of how participants do this requires some groundwork.

Definition 19 (Final Observable Actions and Final Observable Nodes)**.** An action a is *final observable action* for a participant p if there is at least one valid enactment trace for the choreography in which a is the latest observable action (see 15) by p. An AWM node n is a *final observable node* for a participant p if it represents the firing of a participant-activated event associated with the performing of a final observable action for p.

It is important to notice that in some enactment a participant might observe multiple final observable actions being performed, as well as one final observable action being performed multiple times. For instance, in the running example (see e.g. Figure 24) the participants p_2 and p_3 have two final actions, namely the dispatching performed when enactment the message exchange activities mex_3 and mex_4. A choreography in which one final action can be performed multiple times in an enactment is the one shown in Figure 25. In that choreography, p_2 has only one final action, namely $[mex_1]\ p_1 \xrightarrow{m_1} p_2$ which, depending on the decisions taken by p_1, can be performed any number of times.

The identification of the final observable nodes for a participant p is easily achieved by adapting the classic flow-analysis method for calculating *reaching*

definitions. In imperative programming languages, a reaching definition for an instruction i is an assign instruction i' affecting a certain variable x that may reach i without an intervening assignment to x [Aho07]. For reasons of brevity we omit the details of the reaching definitions algorithm, which can be found for example in [MR90]. The intuition of the reaching definitions algorithm is the following: each node n representing an instruction in the program is annotated with pointers to those assignment instructions that are reaching definitions to n for some variable in the program. These pointers to reaching definitions are grouped by the variables they affect. Depending on the CFG of the program, one node may be annotated with multiple reaching definitions for one certain variable.

The problem of identifying the final action nodes of a participant p can be mapped to the concept of reaching definitions as follows. If p is an acting participant of n, it counts as an assignment of the value n to the "variable" named p. All the reaching definitions $n_1, \ldots, n_m$ for p to an end node n_e are the final observable nodes of p in all the enactments ending with n_e. Naturally, it is possible that the various end nodes are associated with different sets of reaching definitions. The overall set of final observable nodes for p is the union of the reaching definitions for p on all the end nodes of the AWM. In the reminder, the function that returns the set of final observable nodes of the participant p in an AWM is denoted by:

$$finalObservableNodes\,(p)$$

As previously mentioned, a participant might observe multiple final actions being performed in one enactment. One of those final actions will *actually* be the last one that the participant observes in the enactment (and therefore the one that signifies the end of that participant's role), but how can the participant know which one? Consider again the choreography shown in Figure 27: the only final observable node of the participant p_2 is $\texttt{[}mex_1\texttt{]}\ p_1 \rightarrow m_1\ \texttt{to}\ p_2$, but the dispatching by p_1 of a message of type m_1 to p_2 may happen any number of times in an enactment, depending on the decisions of p_1. In such a case, in order for a participant to imply that its role is over, the choreography must be specified so that the participant is aware of the firing of *canary events*, i.e. events whose firing guarantees that no final actions for the participant can be further performed.

Definition 20 (Canary Events and Canary Nodes)**.** The event e, the firing of which is represented in the AWM by the node n, is a *canary event* for the participant p if no final observable node of p is reachable from n and p is immediately- or eventually-aware of the firing of e. Put formally:

$$\text{🐤}\,(n,p) := \big(\nexists\, n' \in finalObservableNodes\,(p) : n' \text{ is reachable from } n\big) \wedge \big(\mathrm{AW}_{\mathrm{fired}}\,(n,p) \in \{\mathrm{ia}, \mathrm{ea}\}\big)$$

An AWM node that represents a canary event is a *canary node*.

In other words, the observation of the firing of a canary event by a participant guarantees that no actions observable by that participant lie on any of the paths connecting the respective canary node and any end nodes.[12] Most canary events are reactive events (see Definition 11), i.e. those events that are triggered "on-cascade" by the performing of some action.

As discussed in Section 6.2, because of parallel activities, an enactment may concurrently traverse multiple paths on the AWM. This means that a participant might need to observe multiple canary events before it can deduce its role is over.

Definition 21 (Sufficient Canary Events and Nodes)**.** A set of canary events $e_1, \ldots, e_m$ of the participant p is *sufficient for the event* e if, in any possible enactment, after the firing of e the participant p can observe at least one of $e_1, \ldots, e_m$. Similarly, the set of canary nodes $n_1, \ldots, n_m$ of the participant p is *sufficient for the node* n if every path connecting n with any of the end nodes of the AWM contains at least one of $n_1, \ldots, n_m$.

That is, a set of canary events of a participant is sufficient for a certain event e if, no matter how the enactment evolves after the firing of e, at least one of those canary events will be observed by that participant. This provides a way of verifying the "know when the role is over"-requirement for strong realizability: *a participant knows when its role is over if, for each of its final observable actions, there is a sufficient set of canary events.*

6.3.1 The Bird-Watching Algorithm

The Bird-Watching Algorithm (BWA) presented in Figure 28 checks if a participant p has sufficient canary events for each of its final observable actions.[13] Similarly to the AAA presented in Section 5.2, the BWA presented in Figure 28 is a flow-analysis algorithm. The idea is to use a fixed-point algorithm instead of an enumeration of the paths, possibly infinitely many due to loops in the AWMs, that connect final observable- and end nodes.

The BWA is run once for each of the choreography's participants. As opposed to the forward direction of the AAA (see Section 5), in the BWA the paths in the AWM are traversed by following the control-flow edges *backwards*.

[12] The term "canary event" is clearly a reference to the distasteful mining practices (hopefully) of yore. In stark disregard of animal life, miners brought along caged canaries down the shafts. The canaries were (ab)used as low-tech gas detectors, as they would die sooner than the miners in the presence of toxic gases such as carbon monoxide, methane or carbon dioxide. Thus, the death of a canary meant that the shaft had to be fled really, really quickly. In our realizability analysis, the firing of canary events signals the end of a participant's role. We like to think that our canaries deliver their signal by singing, instead of by perishing. No innocent birds need ever shed their lives on the altar of choreography realizability.

[13] The name "Bird-Watching Algorithm" is an admittedly bad pun on the fact that the participants need to be able to "spot" canaries.

Bird-Watching Algorithm

```
1  /* awm is the AWM of the choreography */
2  /* p is the participant being checked */
3  /* when the BWA terminates, each node of awm reachable from */
4  /* the start node is annotated with input- and output values */
5  function birdingWatchingAlgorithm (awareness model awm,
6       participant p) returns void {
7    /* retrieve the start node of the AWM */
8    node start := startNode(awm);
9
10   /* initialization; 𝕀𝕟(awm) is the set of nodes in the AWM awm */
11   foreach node n in 𝕀𝕟(awm) reachable from start do
12     /* "conservative" initialization */
13     SC_in(n) := CANNOT;
14     SC_out(n) := f^👁(p, n, SC_in(n));
15   end foreach
16
17   /* iteration until a fixed-point is reached */
18   repeat
19     foreach node n in 𝕀𝕟(awm) do
20       /* safe-approximation of the outputs of n's successors */
21       /* see Section 6.3.2 */
22       SC_in(n) := ⅄^👁_{n' ∈ succ(n)} (SC_out(n'));
23       /* update the output-value of n, see Section 6.3.3 */
24       /* the identification of canary events performed */
25       /* during f^👁 can be cached for reuse */
26       SC_out(n) := f^👁(p, n, SC_in(n));
27     end foreach
28   until no SC_out(n) changes
29 }
```

Figure 28: Pseudo-code of the Bird-Watching Algorithm.

In the run of the BWA for a participant p, each node n is associated in the BWA with input and output values, denoted by in this case by $\mathrm{SC}_{\mathrm{in}}(n)$ and $\mathrm{SC}_{\mathrm{out}}(n)$, respectively. (SC stands for "Spots Canaries.") $\mathrm{SC}_{\mathrm{in}}(n)$ represents the safe-approximation of the output-values of the *successors* of the node n, and $\mathrm{SC}_{\mathrm{out}}(n)$ symbolically represents the capability of p to spot the end of its role after the traversal of n. The possible values of $\mathrm{SC}_{\mathrm{in}}(n)$ and $\mathrm{SC}_{\mathrm{out}}(n)$ are the following:

- CAN: n is a canary node of p, or there are sufficient canary events of p for the node n (see Definition 21);
- CANNOT: p does not have sufficient canary events for the node n.

Ideally, when the BWA reaches a fixed-point, *all the final observable nodes* of p have CAN as output-value. In fact, if a final observable node n has an output-value different than CAN, it means that there is some path connecting n and some end node n_e in which p cannot spot canary events that let it imply the end of its role. Therefore, a final observable node of p not annotated with CAN as output-value in the fixed-point reached by the BWA is a symptom of a realizability defect. In the AWMs, this is represented by means of the following awareness constraints attached to each of the final observable nodes $n_{f_1}, \ldots, n_{f_m}$ of p:

$$\mathrm{SC}_{\mathrm{out}}(n_{f_{i\in[1,m]}}) = \mathrm{CAN}$$

Figure 29 shows the input- and output-values associated with the nodes in the fixed-point obtained by applying the BWA on the AWM of the running example, as well as the awareness constraints that verify for p_3 the "know when the role is over"-requirement. (For completeness, the same awareness constraints are also reported in the AWM of the running example shown in Figure 24.) The participant p_3 has two final observable nodes, namely:

- `[`mex_3`]` $p_3 \rightarrow m_3$ `to` p_1`,` p_2
- `[`mex_4`]` $p_1 \rightarrow m_4$ `to` p_2`,` p_3

Each of them is associated with an awareness constraint that verifies if, when the BWA reaches the fixed-point, the output value associated to the node is CAN. This is the case of the `[`mex_4`]` $p_1 \rightarrow m_4$ `to` p_2`,` p_3 node. There is only one path connecting it to the only reachable end node, namely `chor [`$chor_1$`] end`, and along that path each node is a canary node. (This is a special case, due to the fact that no further actions are performed by any participant along that path.) The awareness constraint associated to the node `[`mex_3`]` $p_3 \rightarrow m_3$ `to` p_1`,` p_2 is instead unsatisfied. The reason is that the node `[`mex_3`]` $p_3 \rightarrow m_3$ `to` p_1`,` p_2 is a final observable node, and if p_1 decides to execute the $choice_1 either$ branch, there are no canary nodes for p_3 on the path leading to `chor [`$chor_1$`] err` e_2. Therefore, in enactments in which p_1 decides to enact the $choice_1 either$ branch, p_3 cannot "know when the role is over."

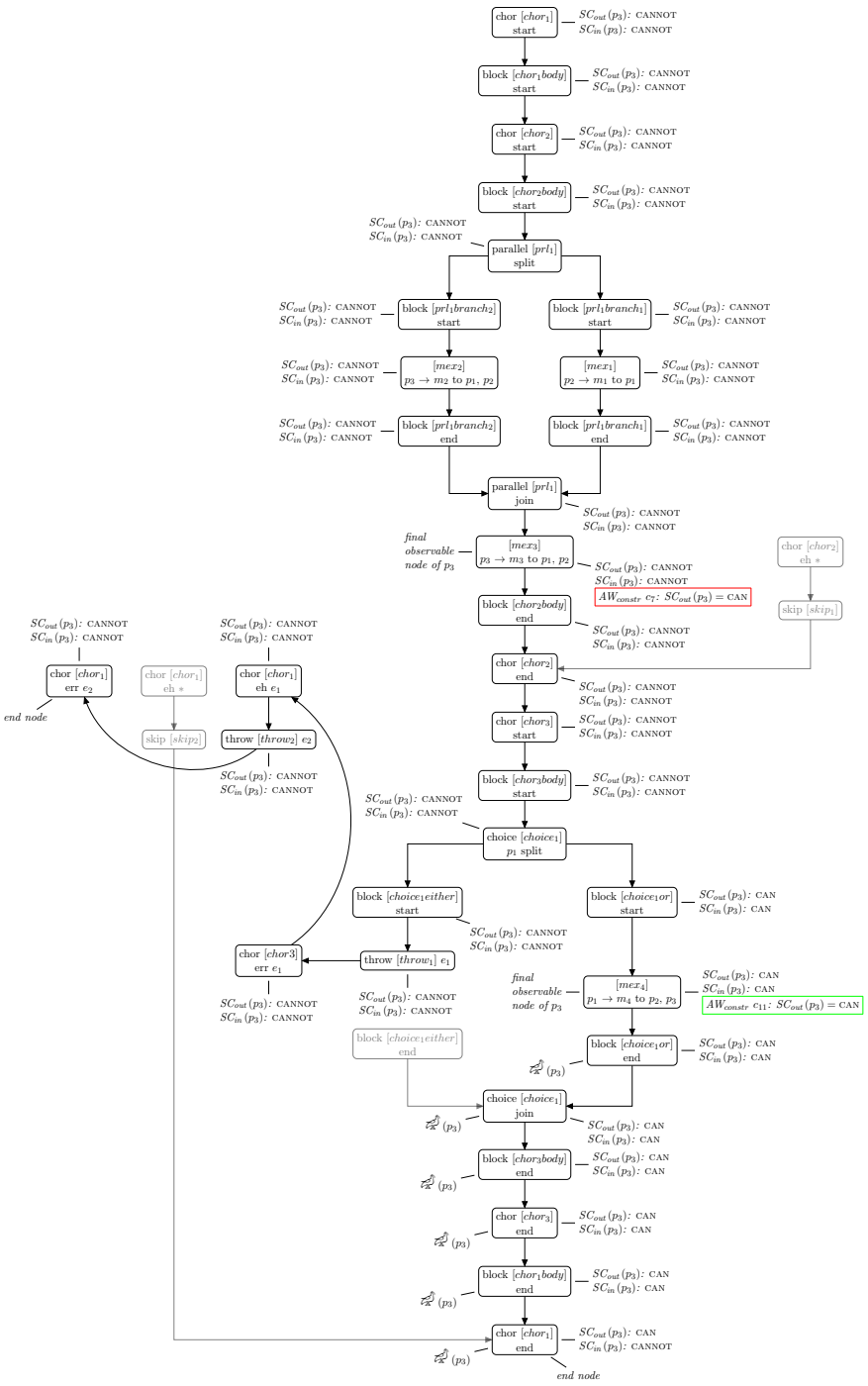


Figure 29: The fixed-point reached by the BWA on the AWM shown in Figure 22 for the participant p_3, with additionally the "know when the role is over" awareness constraints for p_3 (boxed in green when satisfied, red otherwise) and the highlighting of canary- and final observable nodes.

6.3.2 Merge of Input Values

The merge of output values of multiple successors is handled by the function $\curlywedge^{\text{👁}}$ defined in Figure 30.[14] The function $\curlywedge^{\text{👁}}$ takes as arguments one or more output-values of nodes, and returns CAN if all the values are CAN, and CANNOT otherwise.

$$v_{1,\ldots,m} \in \{\text{CAN}, \text{CANNOT}\}$$

$$\curlywedge^{\text{👁}}(v_1, \ldots, v_m) := \begin{cases} \text{CAN} & \text{if } \forall\, i \in [1, m] : v_i = \text{CAN} \\ \text{CANNOT} & \text{otherwise} \end{cases}$$

Figure 30: Recursive definition of the function $\curlywedge^{\text{👁}}$ which merges output values of the predecessors of a node in the BWA.

6.3.3 Update of Output Values

The output-value of nodes is calculated using the function $f^{\text{👁}}$ defined in Figure 31. The predicate "n is canary node of p" is defined in Definition 20. The set $\mathbb{N}^{\text{end}}(chor)$ is the set of end nodes of the choreography $chor$. The function first $(\mathbf{A})$ returns the AWM node generated from the activity $\mathbf{A}$ and labeled as "first." Finally, and the function $sourceActivity(n)$ returns the activity from which the AWM node n has been generated (see Section 4.1).

The outcome of $f^{\text{👁}}(p, n, v)$ is CAN in the following three cases:

- n is a canary node for p;
- v is CAN (due to the "otherwise" case);
- n is a `parallel split` node, and every end node reachable from n is also reachable from the first node of one or more branches of the parallel activity that have CAN as output value.

The rationale of the last case is the following. Recall that, since there are possible multiple end nodes for the choreography, it can be the case that more than one end node is reachable from the `parallel split` node, i.e. in the case that one or more exceptions can propagate from the branches of the parallel activity. Since all the branches of a parallel activity are concurrently enacted, it is not necessary for the participant p to observe along each branch

[14] Apologies are extended to those readers who, like the authors, find the 👁 symbol somewhat creepy. But an eye symbol is just *too* appropriate in the context of the Bird-Watching Algorithm.

$$v \in \{\text{CAN}, \text{CANNOT}\}$$

$$f^{\text{👁}}(p,n,v) := \begin{cases} \text{CAN} & \text{if } \not\text{👁}(n,p) \\ \\ \text{CAN} & \text{if } \neg \not\text{👁}(n,p) \ \wedge \ n = \texttt{parallel split} \wedge \\ & \quad \mathit{sourceActivity}(n) = \texttt{parallel do } \mathbf{A}_1 \texttt{ and ... and } \mathbf{A}_m \wedge \\ & \quad \forall\, n_e \in \mathbb{N}^{\text{end}}(\mathit{chor}) \ \exists\, i \in [1,n] : n_e \text{ reachable from first } (\mathbf{A}_i) \wedge \\ & \quad \text{SC}_{\text{out}}(\text{first}(\mathbf{A}_i)) = \text{CAN} \\ \\ v & \text{otherwise} \end{cases}$$

Figure 31: Definition of the function $f^{\text{👁}}$ which updates output values in the BWA.

canary events for each end node that can be reached from the `parallel split`. Instead, given one end node, it is sufficient that for p to observe a canary event – and therefore deduce the end of its role – along *any* of them. In the definition of the function $f^{\text{👁}}$, this is specified by checking that for each end node reachable from the `parallel split` node, the parallel activity has at least one branch whose "first node" is annotated with CAN as output-value (i.e. canary events will be spotted along that branch), and that the end node is reachable from it.

6.3.4 Computational Complexity of the BWA

The BWA is a fixed-point flow algorithm that is performed once per participant in the choreography. Each invocation of the merge function $\text{人}^{\text{👁}}$ has complexity linear to 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, so we can approximate the evaluation of $\text{人}^{\text{👁}}$ to $\mathcal{O}(1)$.

The definition of the update-function $f^{\text{👁}}$ is case-based. The upper-bound complexity of evaluation of its first case, i.e. if the node n is a canary node, is linear with with the amount of final observable nodes (see Definition 20). The same argument we have made for $\text{人}^{\text{👁}}$ can be applied here, and the evaluation of this case is approximated to $\mathcal{O}(1)$. The second case depends on the amount of end nodes of the choreography and the number of branches of a parallel activity. The amount of end nodes of a choreography and branches of parallel activities is usually negligible with respect to the overall amount of activities specified, therefore we also approximate the evaluation of this case to $\mathcal{O}(1)$.

Notice that a reachability analysis has already performed when performing the AAA algorithm (see Section 5.2.4), and since the AWM nodes and edges have not since changed, its outcome can be reused.

The identification of the final observable nodes is performed by "repurposing" the classic flow-analysis algorithm to calculate reaching definitions, and its upper-bound complexity is thus $\left(|\mathbb{N}\left(awm\right)|^2\right)$. The final observable nodes can be calculated only once per participant, and the outcome of this analysis can be used in both the BWA and the identification of canary nodes performed in $f^{\text{👁}}$.

Since we have estimated the upper-bound complexity of the function $f^{\text{👁}}$ to $\mathcal{O}\left(1\right)$, the upper-bound complexity of the BWA of the initialization phase is:

$$\mathcal{O}\left(\mathbb{N}\left(awm\right)\right)$$

To reach the fixed-point, the BWA performes $\mathcal{O}\left(|\mathbb{N}\left(awm\right)|^2\right)$ iterations, each one with complexity of $\mathcal{O}\left(1\right)$. Recall that both $\lambda^{\text{👁}}$ and $f^{\text{👁}}$ have upper-bound complexity estimated to $\mathcal{O}\left(1\right)$. Therefore, the overall upper-bound complexity of a run of the BWA for a given participant is:

$$\mathcal{O}\left(|\mathbb{N}\left(awm\right)| + |\mathbb{N}\left(awm\right)|^2\right) \approx \mathcal{O}\left(|\mathbb{N}\left(awm\right)|^2\right)$$

Since the BWA is performed once per participant, the overall upper-bound complexity of all its runs in the scope of one realizability analysis is the following:

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

7 Discussion

To the best of our knowledge, the concept of awareness has been previously adopted to investigate the realizability of interaction choreographies only in our previous work [MCvdHP08] and by Desai and Singh in [DS08]. Both previous approaches assume synchronous communication between the participants and, as a consequence, the notion of awareness therein explored is on a "yes/no" basis, i.e. a participant is either (immediately) aware of an event, or unaware of it. In this work have reworked the notion of awareness to accommodate asynchronous messaging. Since participants involved in an action may become aware of its completion at different points in time (e.g. the sender and the recipients of one message exchange), we have introduced the novel concept of "eventual awareness," i.e. that a participant will become aware at some later time that an event has been or could be fired.

7.1 Overall Upper-Bound Computational Complexity of the Awareness-Based Realizability Analysis

The diagram in Figure 32 combines information on the upper-bound complexity of the Awareness Annotation Algorithm (Section 5.2.4), the Bird-Watching

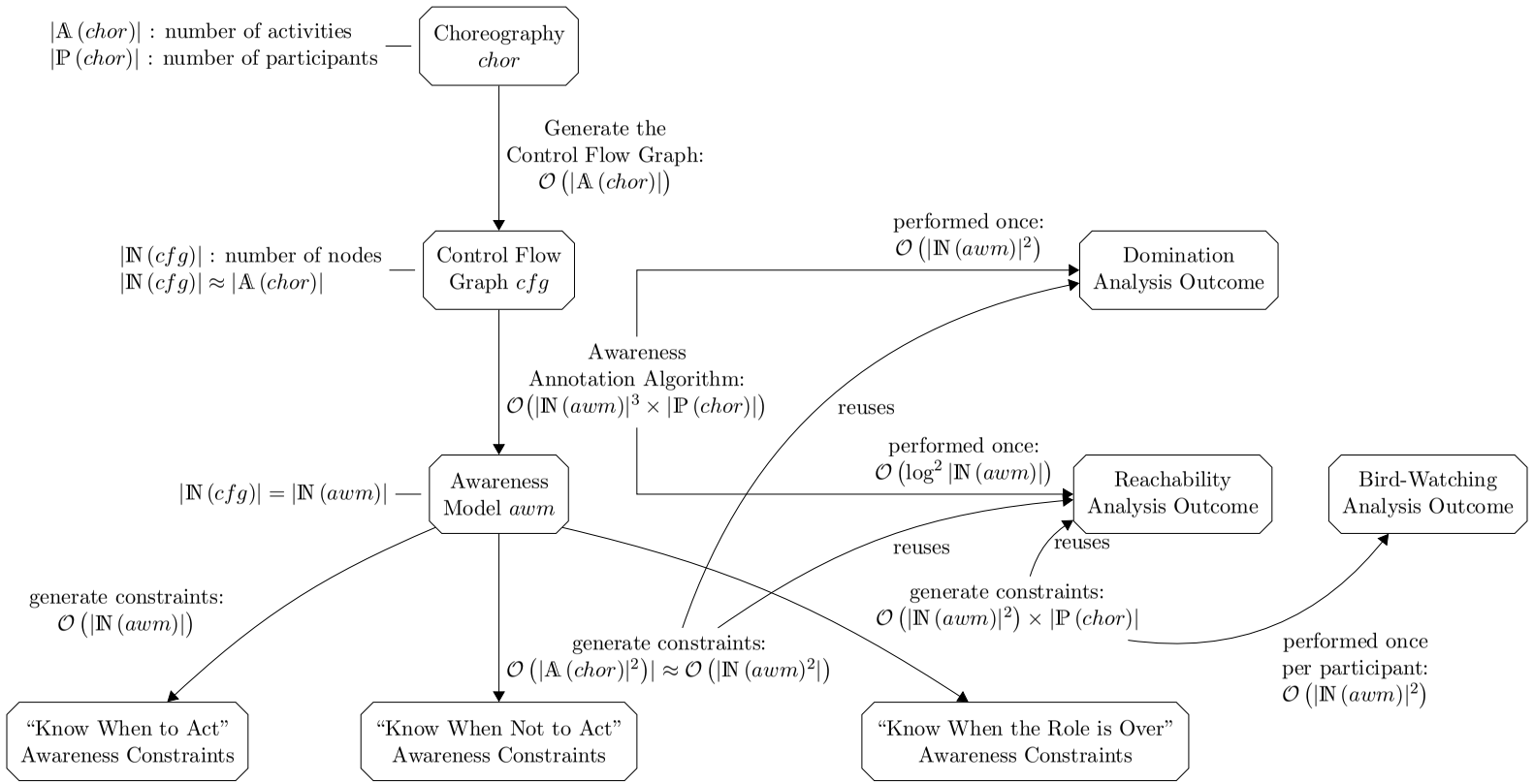


Figure 32: Diagram that summarizes the awareness-based realizability analysis, and correlates the upper-bound complexity computational complexity of its different parts.

Algorithm (see Section 6.3.4), and the unnamed algorithm presented in Section 6.2.1 with information on the complexity of generating the CFG and generating and verifying the awareness constraints that is discussed in the reminder of the present section.

The complexity of generating a CFG from a choreography is linear with respect to the number of activities therein defined (see Section 4.1). As discussed in Section 5.2.4, the amount of nodes in the CFG is comparable (but strictly bigger) than the amount of activities specified in the choreography.

The upper-bound complexity of the AAA was estimated in Section 5.2.4 to the following:

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

The notation $|\mathbb{N}\,(awm)|$ denotes the amount of nodes in the AWM awm. Since the upper-bound complexity above 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.

Once generated, the evaluation of awareness constraints is mostly a matter of looking up a pre-calculated value annotated on the AWM nodes, and it is therefore negligible. The generation of the awareness constraints, on the other hand, is significant in terms of computational complexity, and depends on the type of awareness constraints taken into account. In the case of the awareness constraints that realize the "know when to act"-requirement, it is a matter of scanning all the nodes of the AWM and looking up their acting participants; therefore, it has an upper-bound computational complexity of $\mathcal{O}\big(|\mathbb{N}\,(awm)|\big)$. The upper-bound complexity of the algorithm to generate the "know when to act" has been estimated in Section 6.2.2 to:

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

Finally, the generation of the awareness constraints for verifying the "know when the role is over"-requirement consists in performing the BWA once for every participant in the choreography. Its upper-bound complexity has already been estimated in Section 6.3.4 to:

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

Combining the upper-bound computational complexity of all its parts, the overall upper-bound computational complexity of the awareness-based realizability analysis presented in this work is shown in Figure 33. The final outcome is approximated keeping into account that (1) 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 AWM (as discussed earlier in this section).

$\mathcal{O}\left(\lvert\mathbb{A}\left(chor\right)\rvert\right)+$	(Generation CFG)
$\mathcal{O}\left(\lvert\mathbb{N}\left(awm\right)\rvert^2\right)+$	(Domination Analysis)
$\mathcal{O}\left(\log^2\lvert\mathbb{N}\left(awm\right)\rvert\right)+$	(Reachability Analysis)
$\mathcal{O}\left(\lvert\mathbb{N}\left(awm\right)\rvert^3\times\lvert\mathbb{P}\left(chor\right)\rvert\right)+$	(Awareness Annotation Algorithm)
$\mathcal{O}\left(\lvert\mathbb{N}\left(awm\right)\rvert\right)+$	(Generation "know when to act" constraints)
$\mathcal{O}\left(\lvert\mathbb{A}\left(chor\right)\rvert^2\right)+$	(Generation "know when not to act" constraints)
$\mathcal{O}\left(\lvert\mathbb{N}\left(awm\right)\rvert^2\times\lvert\mathbb{P}\left(chor\right)\rvert\right)\approx$	(Generation "know when the role is over" constraints)
$\mathcal{O}\left(\lvert\mathbb{N}\left(awm\right)\rvert^3\times\lvert\mathbb{P}\left(chor\right)\rvert\right)$	(Overall Upper-Bound Complexity)

Figure 33: Overall upper-bound computational complexity of the awareness-based realizability analysis of ChorTex choreographies.

Given the variability of realizability definitions in the state of the art (see Section 3), and their dependency on particular choreography modeling languages (see Section 3.1), it makes little sense to compare realizability analysis methods devised for different languages. Nevertheless, the computational complexity of our method compares very favorably with the methods based on automata [FBS05, KP06, MFEH07, HB10] or Petri-Nets [DW07, LW09], which resort to PSPACE-complete language-equivalence checks (see also [LW11]). The computational complexity of our realizability analysis method comes mostly from the adoption of eventual awareness. 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.

7.2 A Case Against Exception Handling in Choreographies

The awareness constraints realizing the "know when not to act"-requirement make a compelling case *against* the adoption in choreography modeling languages that allow asynchronous communication of exception handling constructs that are closely inspired from Object-Oriented (OO) programming languages. The prevention of violations caused by the throwing of exceptions imposes strong limitations on the design of ChorTex choreographies. In fact, exception throwing is "safe" 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 5). Since all participants that may act concurrently to the throwing of an exception must be immediately aware of the latter ("know when not to act"-requirement, see Section 6.2), 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.

Interestingly, a similar observation is found in the BPMN v2.0 Choreography specification [OMG11, p. 344] with respect to termination end events:

> "[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 is not surprising, given the fact that a BPMN v2.0 termination end event has the same effect of the throwing of an exception that terminates an entire ChorTex choreography. Exception handling and termination events are constructs that work well in service orchestrations, where all the activities are executed by the same actor (e.g. the orchestration engine).

Given the tight relationship between service orchestrations and choreographies, we understand how tempting it is to "port" service orchestration constructs to service choreographies. After all, similar constructs in service orchestrations and choreographies intuitively simplifies the projection of the roles [YZCQ07] and softens the learning curve for the modelers. However, the distributed nature of service choreographies most likely requires a radical re-thinking of how error handling is performed, see e.g. [KWL09].

Acknowledgments

The authors extend many thanks to Oliver Kopp for the precious feedback.

References

[AEY03] Rajeev Alur, Kousha Etessami, and Mihalis Yannakakis. Inference of message sequence charts. *IEEE Trans. Software Eng.*, 29(7):623–633, 2003.

[AEY05] Rajeev Alur, Kousha Etessami, and Mihalis Yannakakis. Realizability and verification of MSC graphs. *Theor. Comput. Sci.*, 331(1):97–114, 2005.

[Aho07] A.V. Aho. *Compilers: principles, techniques, & tools*. Addison-Wesley series in computer science. Pearson/Addison Wesley, 2007.

[All70] Frances E. Allen. Control flow analysis. *SIGPLAN Not.*, 5:1–19, July 1970.

[AtHEvdA06] Michael Adams, Arthur H. M. ter Hofstede, David Edmond, and Wil M. P. van der Aalst. Worklets: A service-oriented implementation of dynamic flexibility in workflows. In Robert Meersman and Zahir Tari, editors, *OTM Conferences (1)*, volume 4275 of *Lecture Notes in Computer Science*, pages 291–308. Springer, 2006.

[BCC+06] Don Box, Erik Christensen, Francisco Curbera, Donald Ferguson, Jeffrey Frey, Marc Hadley, Chris Kaler, David Langworthy, Frank Leymann, Brad Lovering, Steve Lucco, Steve Millet, Nirmal Mukhi, Mark Nottingham, David Orchard, John Shewchuk, Eugéne Sindambiwe, Tony Storey, Sanjiva Weerawarana, and Steve Winkler. Web Services Addressing (WS-Addressing) Version 1.0. W3C Recommendation, W3C, May 2006.

[BF07] Tevfik Bultan and Xiang Fu. Specification of realizable service conversations using collaboration diagrams. In *SOCA*, pages 122–132. IEEE Computer Society, 2007.

[BF08] Tevfik Bultan and Xiang Fu. Choreography modeling and analysis with collaboration diagrams. *IEEE Data Eng. Bull.*, 31(3):27–30, 2008.

[BFS07] Tevfik Bultan, Xiang Fu, and Jianwen Su. Analyzing conversations: Realizability, synchronizability, and verification. In Luciano Baresi and Elisabetta Di Nitto, editors, *Test and Analysis of Web Services*, pages 57–85. Springer, 2007.

[BH10] Benedikt Bollig and Loïc Hélouët. Realizability of dynamic MSC languages. In Farid M. Ablayev and Ernst W. Mayr, editors, *CSR*, volume 6072 of *Lecture Notes in Computer Science*, pages 48–59. Springer, 2010.

[BIM95] Bard Bloom, Sorin Istrail, and Albert R. Meyer. Bisimulation can't be traced. *J. ACM*, 42(1):232–268, 1995.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008.

[BTZ12] Massimo Bartoletti, Emilio Tuosto, and Roberto Zunino. On the realizability of contracts in dishonest systems. *CoRR*, abs/1201.6188, 2012.

[Cal88] David Callahan. The program summary graph and flow-sensitive interprocedural data flow analysis. In *PLDI*, pages 47–56, 1988.

[CHK01] Keith D. Cooper, Timothy J. Harvey, and Ken Kenned. A simple, fast dominance algorithm. *Software – Practice and Experience*, 4(1):1–10, 2001.

[DDC09] Hywel R. Dunn-Davies and Jim Cunningham. Verifying realizability and reachability in recursive interaction protocol specifications. In Carles Sierra, Cristiano Castelfranchi, Keith S.

Decker, and Jaime Simão Sichman, editors, *AAMAS (2)*, pages 1233–1234. IFAAMAS, 2009.

[Dec09] Gero Decker. *Design and Analysis of Process Choreographies*. PhD thesis, Hasso Plattner Institute, 2009.

[DKB08] Gero Decker, Oliver Kopp, and Alistair P. Barros. An introduction to service choreographies (Servicechoreographien - eine Einführung). *it - Information Technology*, 50(2):122–127, 2008.

[DKLW07] Gero Decker, Oliver Kopp, Frank Leymann, and Mathias Weske. BPEL4Chor: Extending BPEL for modeling choreographies. In *ICWS*, pages 296–303. IEEE Computer Society, 2007.

[DKLW09] Gero Decker, Oliver Kopp, Frank Leymann, and Mathias Weske. Interacting services: From specification to execution. *Data Knowl. Eng.*, 68(10):946–972, 2009.

[DS08] Nirmit Desai and Munindar P. Singh. On the enactability of business protocols. In Dieter Fox and Carla P. Gomes, editors, *AAAI*, pages 1126–1131. AAAI Press, 2008.

[DW07] Gero Decker and Mathias Weske. Local enforceability in interaction Petri Nets. In Gustavo Alonso, Peter Dadam, and Michael Rosemann, editors, *BPM*, volume 4714 of *Lecture Notes in Computer Science*, pages 305–319. Springer, 2007.

[DW11] Gero Decker and Mathias Weske. Interaction-centric modeling of process choreographies. *Inf. Syst.*, 36(2):292–312, 2011.

[FBS05] Xiang Fu, Tevfik Bultan, and Jianwen Su. Realizability of conversation protocols with message contents. *Int. J. Web Service Res.*, 2(4):68–93, 2005.

[FPD+09] Paul Fremantle, Sanjay Patil, Doug Davis, Anish Karmarkar, Gilbert Pilz, Steve Winkler, and Ümit Yalçinalp. Web Services Reliable Messaging (WS-ReliableMessaging) Version 1.2. OASIS Standard, OASIS, February 2009.

[HB10] Sylvain Hallé and Tevfik Bultan. Realizability analysis for message-based interactions using shared-state projections. In Gruia-Catalin Roman and Kevin J. Sullivan, editors, *SIGSOFT FSE*, pages 27–36. ACM, 2010.

[KP06] Raman Kazhamiakin and Marco Pistore. Analysis of realizability conditions for web service choreographies. In Elie Najm,

Jean-François Pradat-Peyre, and Véronique Donzeau-Gouge, editors, *FORTE*, volume 4229 of *Lecture Notes in Computer Science*, pages 61–76. Springer, 2006.

[KWL09] Oliver Kopp, Matthias Wieland, and Frank Leymann. Towards choreography transactions. In *ZEUS*, pages 49–54, 2009.

[Loh08] Niels Lohmann. Correcting deadlocking service choreographies using a simulation-based graph edit distance. In Marlon Dumas, Manfred Reichert, and Ming-Chien Shan, editors, *BPM*, volume 5240 of *Lecture Notes in Computer Science*, pages 132–147. Springer, 2008.

[Lou97] Kenneth C. Louden. *Compiler Construction: Principles and Practice*. PWS Publishing Co., Boston, MA, USA, 1997.

[LW09] Niels Lohmann and Karsten Wolf. Realizability is controllability. In Cosimo Laneve and Jianwen Su, editors, *WS-FM*, volume 6194 of *Lecture Notes in Computer Science*, pages 110–127. Springer, 2009.

[LW11] Niels Lohmann and Karsten Wolf. Decidability results for choreography realization. In Gerti Kappel, Zakaria Maamar, and Hamid R. Motahari-Nezhad, editors, *ICSOC*, volume 7084 of *Lecture Notes in Computer Science*, pages 92–107. Springer, 2011.

[MCvdHP08] Michele Mancioppi, Manuel Carro, Willem-Jan van den Heuvel, and Mike P. Papazoglou. Sound multi-party business protocols for service networks. In Athman Bouguettaya, Ingolf Krüger, and Tiziana Margaria, editors, *ICSOC*, volume 5364 of *Lecture Notes in Computer Science*, pages 302–316, 2008.

[MFEH07] Abdolmajid Mousavi, Behrouz Homayoun Far, Armin Eberlein, and Behrouz Heidari. Strong safe realizability of message sequence chart specifications. In Farhad Arbab and Marjan Sirjani, editors, *FSEN*, volume 4767 of *Lecture Notes in Computer Science*, pages 334–349. Springer, 2007.

[MO04] Markus Müller-Olm. Precise interprocedural dependence analysis of parallel programs. *Theor. Comput. Sci.*, 311(1-3):325–388, 2004.

[MPR+09] Michele Mancioppi, Mikhail Perepletchikov, Caspar Ryan, Willem-Jan van den Heuvel, and Mike P. Papazoglou. Towards a quality model for choreography. In Asit Dan, Frederic Gittler, and Farouk Toumani, editors, *ICSOC/ServiceWave*

Workshops, volume 6275 of *Lecture Notes in Computer Science*, pages 435–444, 2009.

[MR90] Thomas J. Marlowe and Barbara G. Ryder. Properties of data flow frameworks. *Acta Inf.*, 28(2):121–163, 1990.

[Mye81] Eugene W. Myers. A precise interprocedural data flow algorithm. In *POPL*, pages 219–230, 1981.

[OMG11] OMG. Business Process Model and Notation Version 2.0. OMG Specification formal/2011-01-03, Object Management Group, January 2011.

[RHS95] Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, pages 49–61, 1995.

[RS11] Nima Roohi and Gwen Salaün. Realizability and dynamic reconfiguration of Chor specifications. *Informatica*, 2011. To appear.

[SBFZ07] Jianwen Su, Tevfik Bultan, Xiang Fu, and Xiangpeng Zhao. Towards a theory of web service choreographies. In Marlon Dumas and Reiko Heckel, editors, *WS-FM*, volume 4937 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2007.

[SH00] Saurabh Sinha and Mary Jean Harrold. Analysis and testing of programs with exception handling constructs. *IEEE Trans. Software Eng.*, 26(9):849–871, 2000.

[Tar74] Robert Endre Tarjan. Testing flow graph reducibility. *J. Comput. Syst. Sci.*, 9(3):355–365, 1974.

[vG93] Rob J. van Glabbeek. The linear time - branching time spectrum II. In Eike Best, editor, *CONCUR*, volume 715 of *Lecture Notes in Computer Science*, pages 66–81. Springer, 1993.

[W3C05] W3C. Web Services Choreography Description Language Version 1.0. Candidate Recommendation, W3C, November 2005.

[WRRM08] Barbara Weber, Manfred Reichert, and Stefanie Rinderle-Ma. Change patterns and change support features - enhancing flexibility in process-aware information systems. *Data Knowl. Eng.*, 66(3):438–466, 2008.

[YZCQ07] Hongli Yang, Xiangpeng Zhao, Chao Cai, and Zongyan Qiu. Exploring the connection of choreography and orchestration with exception handling and finalization/compensation. In John Derrick and Jüri Vain, editors, *FORTE*, volume 4574 of *Lecture Notes in Computer Science*, pages 81–96. Springer, 2007.