Humberto Nicolás Castejón,
Gregor von Bochmann and Rolv Bræk

Investigating the Realizability of Collaboration-based Service Specifications

Avantel Technical Report

DEPARTMENT OF TELEMATICS, NTNU
ISSN 1503-4097

NTNU
Norwegian University of Science and Technology

Department of Telematics

AVANTEL TECHNICAL REPORT
3/2007

ISSN-no:
1503-4097

TITLE

Investigating the Realizability of Collaboration-based Service Specifications

DATE

2007-09-01

NO. OF PAGES/APPENDICES
87

AUTHOR(S)

Humberto Nicolás Castejón, Gregor von Bochmann and Rolv Bræk

ABSTRACT

This report is concerned with compositional specification of services using UML 2 collaborations, activity and interaction diagrams. It provides formal syntax and semantics for so-called choreography graphs, used to describe the complete behavior of composite collaborations. It then addresses the problem of realizability: given a global specification, can we construct a set of communicating state machines whose joint behavior is precisely the specified one? We approach the problem by looking at how collaboration behaviors may be composed using UML activity diagrams-based choreographies. We classify realizability problems from the point of view of each composition operator, and discuss their nature and possible solutions. This brings a new look at already known problems. We show that given some conditions, some problems can already be detected at an abstract collaboration level, without needing to look into detailed interactions. We present algorithms to detect some of the discussed problems.

KEYWORDS

Service engineering, service-oriented development, model-driven development, UML 2 collaborations, choreography, realizability, race conditions

1 Introduction

For several decades now it has been common practice to specify and design reactive systems in terms of loosely coupled components modeled as communicating state machines [Boc78, Bræ79], using languages such as SDL [IT00] and UML [OMG07]. This has helped to substantially improve quality and modularity, mainly by providing means to define complex, reactive behavior precisely in a way that is understandable to humans and suitable for formal analysis as well as automatic generation of executable code.

However, there is a fundamental problem. In many cases, application/service behavior is not performed by a single component, but by several collaborating components. This is referred to as the "crosscutting" nature of services by different authors [RGG01, FK01, KM03]. Often each component takes part in several different services, so in general, the behavior of services is composed from partial component behaviors, while component behaviors are composed from partial service behaviors. By structuring according to components, the behavior of each individual component can be defined precisely and completely, while the behavior of a service is fragmented. In order to model the global behavior of a service more explicitly one needs an orthogonal view where the collaborative behavior is in focus. Interaction sequences such as MSC [IT99], and UML Sequence diagrams [OMG07] are commonly used for this purpose, but normally only to describe typical/important use cases and not complete behaviors. Normally when using interaction sequences, it is very cumbersome to define all the intended scenarios. In addition, there are problems related to the *realizability* of interaction scenarios, i.e. finding a set of local component behaviors whose joint execution leads precisely to the global behavior specified in the scenarios. The realizability of MSC-based specifications has been extensively studied by different authors (e.g. [AEY00, UKM04, AEY05, BS05]). Conditions for realizability have been proposed for HMSCs [HJ00] and Compositional MSCs [MRW06], as well as restricted classes of HMSCs that are known to be always realizable [GMSZ06]. Some authors have studied pathologies in HMSCs [BAL97, Hél01] that prevent their realization. Other authors have considered realizability notions that allow additional message contents [BM03, GMSZ06].

A promising step forward is to adopt a *collaboration-oriented* approach, where the main structuring units are collaborations. This is made practically possible by the new UML 2 collaboration concept [OMG07]. The underlying ideas, however, date back to before the UML era [RAB+92, RWL96]. Collaborations model the concept of a service very nicely. They define a structure of partial object behaviors, called roles, and enable a precise definition of the service behavior using interaction diagrams, activity diagrams and state machines as explained in [SCKB05, CB06a, CB06b]. They also provide a way to compose services by means of collaboration uses and to bind roles to components. In this way, UML 2 collaborations directly support (crosscutting) service modeling and service composition. As we shall see in the following, this opens many interesting opportunities. Figure 1 illustrates the main models involved in the collaboration oriented approach being discussed in the following:

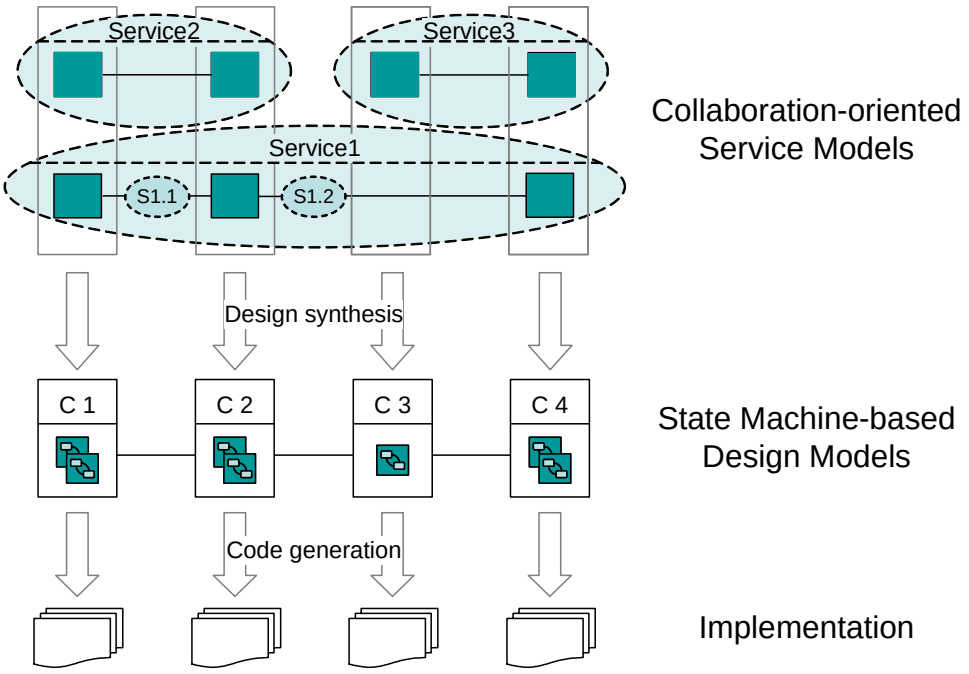

Figure 1: Collaboration oriented development

- Service models are used to formally specify and document services. Collaborations provide a structural framework for these models that can embody both the role behaviors and the interactions between the roles needed to fulfill the service.

- Design models are used to formally specify and document system structure and components realizing the services. They are expressed in terms of communicating state machines, using UML 2 active objects. Each of these will realize one or more collaboration roles.

- Implementations are executable code automatically generated from the design models.

This paper is concerned with the crucial first steps of expressing service models using UML 2 collaborations and deriving well-formed design models expressed as communicating state machines. The ensuing steps from design component models to implementations and dynamic deployment on service platforms can be solved in different ways, see for instance [San00, BM05], and are not discussed further here.

An important property of collaborations is that it is possible and convenient to compose/decompose collaborations structurally into sub-collaborations, by means of collaboration uses. These refer to separately defined collaborations and provide a mechanism for collaboration reuse. In order to define the behavior of collaborations, we have found it useful to distinguish between the behavior of *composite collaborations* and *elementary collaborations* (collaborations that are not further decomposed into sub-collaborations). The elementary collaborations that result from

the decomposition process are often quite simple, reusable and possible to define completely using interaction sequences. Binary collaborations can in many cases be associated with interfaces and their sub-collaborations with features of the interface. The question then is how to define the overall behavior of composite collaborations in terms of sub-collaboration behaviors? In the web service domain this kind of behavior is called "choreography" [Erl05], a term we will use in the following. Several notations may be used to define the choreography of sub-collaborations (i.e. their global execution ordering). We have found UML 2 Activity diagrams a good candidate, as they provide many of the composition operators needed for the purpose. While HMSCs normally describe collection of scenarios, and therefore represent incomplete and existential behavior, our choreographies describe the exact behavior of a service according to the service designer's intentions (i.e. the service should behave exactly as described, no other behaviors are allowed). The local behavior for a given component of the choreography can be obtained by applying the ordering defined by the choreography's activity diagram to the role behaviors bound to the component in question.

We say that a choreography is directly realizable if the joint execution of the local behaviors of all components leads precisely to the global behavior specified by the choreography. Note that some choreographies that are not directly realizable may still be realized by adding extra coordination messages or additional data in messages. We consider these measures as solutions to realization problems, which could be adopted by the designer depending on the context and service domain. Note also that the realizability of a choreography depends not only on the ordering defined by the activity diagram of the choreography, but also on the characteristics of the underlying communication service used for the transmission of messages. Important characteristics of the communication service are the type of transmission channels, and the type and number of input buffers of each component. We assume there is no message loss, and distinguish between channels with out-of-order delivery (i.e. messages sent from a given source to a given destination may be received in a different order than they were sent) and channels with in-order delivery. Components may have either a single input FIFO buffer (i.e. one buffer for all received messages) or separate input FIFO buffers (i.e. one buffer for messages received from each different peer).

In the rest of the paper we study the direct realizability of a choreography from the point of view of the operators used to compose the sub-collaborations. In our discussion we assume that each sub-collaboration of a choreography is directly realizable. Then, for each composition operator (i.e. sequential, alternative, parallel, interruption) we study the problems that can lead to difficulties of realization. We investigate the actual nature of these problems and discuss possible solutions to prevent or remedy them.

1.1 Outline

The paper is structured as follows. In Section 2 the proposed service modeling approach is illustrated with help of an example, and the syntax and semantics of

choreographies is presented in Section 3. The problem of realizability of choreographies is discussed in Section 4. Section 5 presents a set of algorithms to detect some of the realizability problems discussed in the previous section. We finally conclude with Section 6.

2 Service Specification Approach: An Example

We exemplify our service specification approach by means of a simple shuttle service (inspired by a case study from [UKM04]) in which one vehicle transports one passenger at a time between two terminals. Figure 2 depicts this service as a UML 2.0 collaboration. This collaboration identifies three roles, namely P (Passenger), T (Terminal) and V (Vehicle); as well as seven sub-collaborations representing interfaces and features of the service. These sub-collaborations are specified as UML collaboration-uses, whose roles are bound to the *ShuttleService*'s roles (e.g. *BuyTicket*'s role T_{bt} is bound to *ShuttleService*'s role T). For the sake of clarity, in the following we will refer to P, T and V as service-roles, and to $T_{\mathrm{bt}}, T_{\mathrm{d}}$ and the like as sub-roles (of T, P or V). The *ShuttleService*'s sub-collaborations have been identified from the following service requirements. In order to travel, a passenger must buy a ticket at one of the terminals (collaboration-use *BuyTicket*). When this happens, if the vehicle is waiting at the terminal, the departure gate is indicated, and the passenger can enter the vehicle (*EnterVehicle*). The terminal then dispatches the vehicle (*VehDeparture*) and, after arriving at the second terminal (*VehArrival*), the passenger disembarks (*ExitVehicle*). If the vehicle is not at the terminal where the passenger buys the ticket, that terminal requests the vehicle from the other terminal (*ReqVehicle*), which dispatches the vehicle towards the requesting terminal. When the vehicle arrives, the departure gate is displayed and the service continues as explained before.

The complete and exact behavior of each elementary sub-collaboration is described by means of sequence diagrams. Figure 3 shows the sequence diagrams describing *BuyTicket, VehDeparture* and *VehArrival*.

What remains is to specify the overall cross-cutting behavior of the *ShuttleService* collaboration, that is, the choreography describing how its sub-collaborations are ordered and interact. We use UML 2 activity diagrams to describe the choreography of collaborations. They capture the liveness aspects of composite service collaborations by describing the execution order of their sub-collaborations. The choreography for *ShuttleService* is depicted in Fig. 4. Note that we have annotated each activity with a pictorial representation of the collaboration-use the activity refers to.

3 Syntax and Semantics of Choreographies

A choreography graph specifies the full behavior of a composite collaboration by defining the global execution ordering of its sub-collaborations. We consider here choreography graphs represented by means of UML activity diagrams, and assume that the sub-collaborations referred to by the choreography are elementary collab-

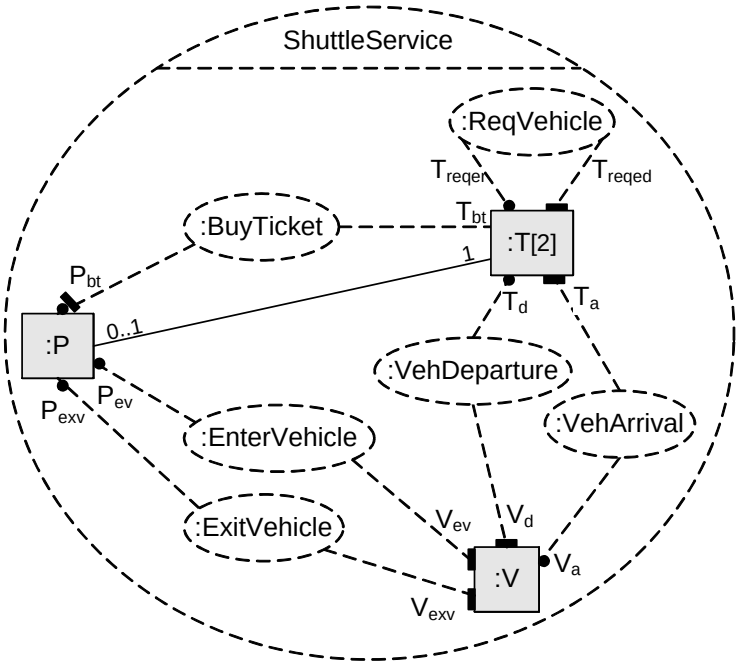

Figure 2: UML collaboration for the *ShuttleService*

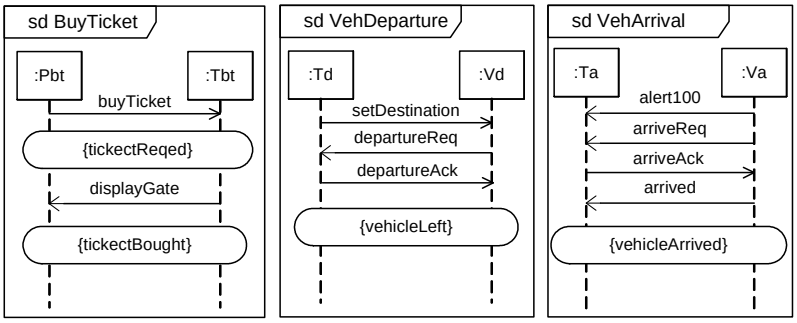

Figure 3: Sequence diagrams describing some elementary collaborations of *Shuttle-Service*

orations whose behavior is described by means of UML sequence diagrams. In the following we describe the syntax and semantics of choreography graphs. Before that we describe the syntax and semantics of sequence diagrams.

3.1 Syntax and Semantics for Sequence Diagrams

We consider here a restricted version of UML 2 sequence diagrams, which we use to specify the behavior of elementary collaborations. We used the syntax proposed in the UML standard, but provide a semantics based on partially ordered sets (posets). In the following we define and provide a semantics to basic sequence diagrams. Thereafter, we focus on composite sequence diagrams.

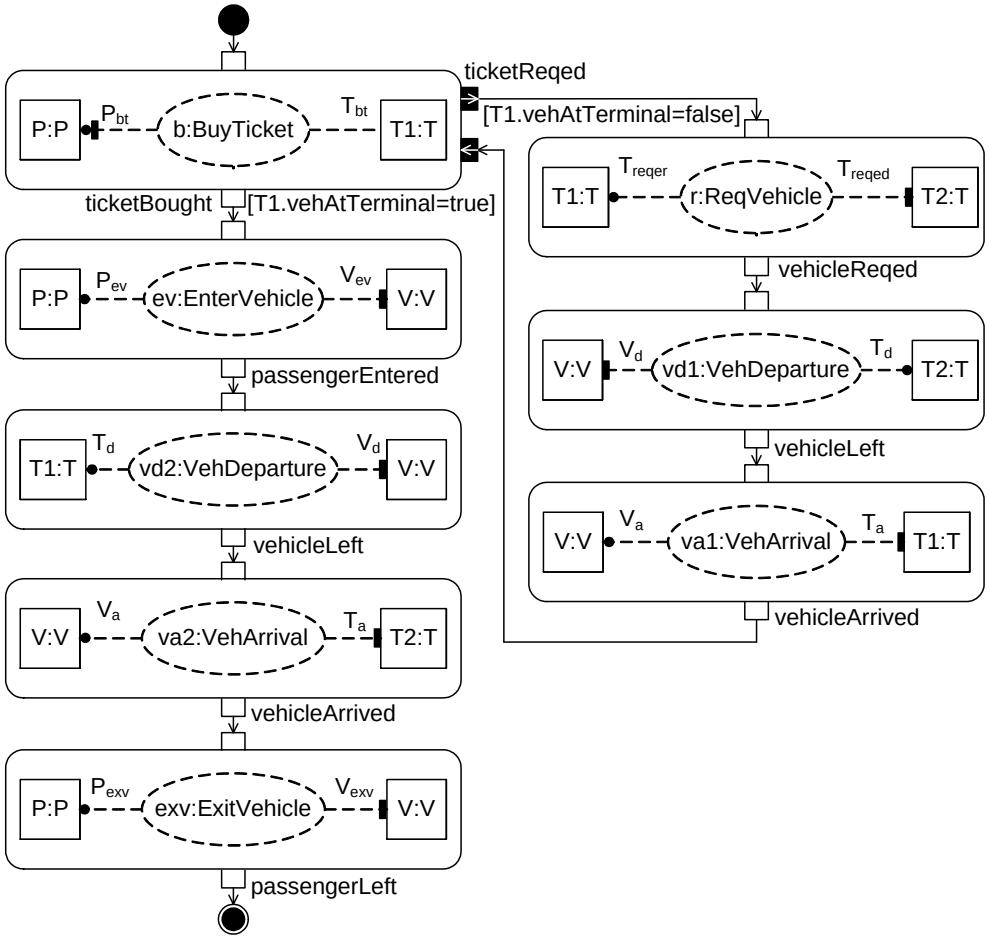

Figure 4: Choreography of *ShuttleService*

Definition 3.1 (**Basic Sequence Diagram**)**.** A basic sequence diagram defines a labeled directed acyclic graph that can be described by a tuple $bSD = (E, <_e, <_m, \mathscr{P}, \mathscr{M}, \Sigma, loc, lbl, rcv, snd)$, where:

- $E = S \cup R \cup \Phi$ is a set of events partitioned into sending events (S), receiving events (R) and predicate events (Φ)
- $\mathscr{P}$ is a set of lifelines
- $\mathscr{M}$ is a set of messages
- $\Sigma = \Sigma_c \cup \Sigma_p$ is a set of communication actions (Σ_c) and predicates (Σ_p). Elements of Σ_c are of the form $\langle !m, p, q\rangle$ or $\langle ?m, p, q\rangle$, with $p, q \in \mathscr{P}, m \in \mathscr{M}$. We read $\langle !m, p, q\rangle$ as "p sends message m to q", and $\langle ?m, p, q\rangle$ as "p receives message m from q"
- $loc : E \to \mathscr{P}$ is a mapping that associates each event with a lifeline
- $lbl : E \to \Sigma_c \cup \Sigma_p$ is a labeling function associating each event with a communicating action or a predicate

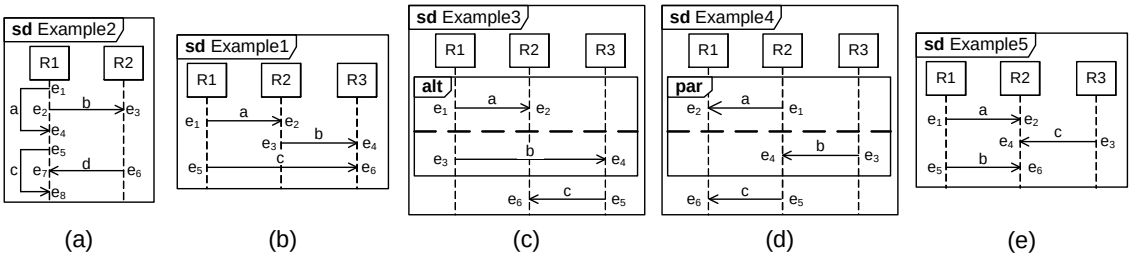

Figure 5: Some basic sequence diagrams (with conflicts)

- $rcv : S \to R$ and $snd : R \to S$ are bijective functions that respectively match each sending event with its corresponding receiving event, and each receiving event with its corresponding sending event. We have $snd \equiv rcv^{-1}$
- $<_e \subseteq E \times E$ is an acyclic relation between events, called the *visual* order, that satisfies:

  $<_e = (\bigcup_{p \in \mathscr{P}} <_p) \cup \{(s, rcv(s)) : s \in S\}$, where for each $p \in \mathscr{P}$, $<_p$ is a total order (i.e. an antisymmetric, transitive and total binary relation) on $E_p = \{e \in E : loc(e) = p\}$ (i.e. events on a lifeline are totally ordered)
- $<_m \subseteq S \times S$ is a total order on sending events, called the *message* order. To construct $<_m$ we imagine a vertical line L aligned with the sequence diagram and project each sending event on L. We say $s_1 <_m s_2$ if s_1 is located on L higher than s_2. To ensure that $<_m$ is a total order, two sending events cannot be drawn on the same imaginary horizontal line.

For any sequence diagram we require the following property:

Property 3.1.1 (**Non-crossing messages**)**.** Messages do not graphically cross each other. That is:

(i) Messages exchanged between different lifelines are represented by horizontal arrows.

(ii) Messages sent by a lifeline $p \in \mathscr{P}$ to itself (*self-receiving* messages) satisfy this condition: $\forall e_1 = \langle !m, p, p \rangle, e_2 = rcv(e_1), \nexists e \text{ such that } e_1 <_p e <_p e_2$[1].

In this property, condition (i) is required to avoid cycles that could lead to causal relations where a sending message is indirectly dependent on its receiving event. Condition (ii) is required because, in general, a specification with an order $\langle !m, p, p \rangle <_p e_1 <_p \ldots e_n <_p rcv(\langle !m, p, p \rangle)$ is not directly realizable. As an example consider the diagram in Fig. 5(a). Here *R1* first sends message a to itself and before receiving it, it sends message b to *R2*. This specification does not satisfy condition (ii) in Property 3.1.1, since we have $e_1 = \langle !a, R1, R1 \rangle <_p e_2 <_p e_4 = \langle ?a, R1, R1 \rangle$. It is easy to see that, in general[1], this order cannot be guaranteed by any directly realized

[1]This order can only be guaranteed if the design is such that the two sending events e_1 and e_2 are performed in an atomic transition.

system, since the reception of message a cannot be controlled, and may therefore happen before the sending of message b. In the lower part of the diagram a similar situation happens between messages c and d.

Both the visual order ($<_e$) and the message order ($<_m$) can be obtained form the graphical representation of the sequence diagram. The visual order captures the order of events on each lifeline, as well as the order induced by message transmissions. The message order captures the vertical order of messages. For example, in Fig. 5(b) the visual order is $<_e = \{(e_1,e_5),(e_2,e_3),(e_4,e_6),(e_1,e_2),\ (e_3,e_4),(e_5,e_6)\}$, while the message order is $<_m = \{(e_1,e_3),(e_3,e_5),(e_1,e_5)\}$, which indirectly describes the vertical order between messages a, b and c.

The ordering of events dictated by the visual order does not necessarily reflect the semantics of the sequence diagram. The visual order may impose ordering between events that cannot be guaranteed in a directly realized system. To describe the semantics of sequence diagrams we use a weaker partial order, called the *causal partial order* (or *causal poset*), which only orders two events if they necessarily happen in that order in any execution of the directly realized system. The causal order takes into account the particularities of the communication architecture to be used in the realized system, in particular whether channels with out-of-order delivery or channels with in-order delivery are to be used.

Definition 3.2. A **causal order for channels with out-of-order (or non-fifo) delivery** ($\prec_{\mathrm{nf}}$) is the reflexive-transitive closure of the immediate precedence relation $<_{\mathrm{nf}}$ (i.e. $\prec_{\mathrm{nf}} = (<_{\mathrm{nf}})^*$), where $e <_{\mathrm{nf}} e'$ if any of the following holds:

- $e \in S \wedge rcv(e) = e'$ (i.e. e' is the receiving event associated to the sending event e)
- $e' \in S \cup \Phi \wedge e <_p e'$, for $p \in \mathscr{P}$ (i.e. e' is a sending event, or a predicate event, and e is a visual predecessor of e' on the same lifeline)

The above definition reflects that in a channel with out-of-order delivery, the order in which messages are received (at a certain input buffer) may not be the same as the order in which message were sent (i.e. messages may overtake each other on the channel). With out-of-order delivery channels, only two classes of event orderings can be guaranteed. First, a receiving event will always happen after its sending event. Second, a system component may always control when to perform a sending event. This means that a sending event s will always happen after any other events that precede s on the same lifeline have been performed.

Definition 3.3. A **causal order for channels with in-order (or fifo) delivery** ($\prec_{\mathrm{f}}$) is the reflexive-transitive closure of the immediate precedence relation $<_{\mathrm{f}}$ (i.e. $\prec_{\mathrm{f}} = (<_{\mathrm{f}})^*$), where $e <_{\mathrm{f}} e'$ if one of the following holds:

- $e <_{\mathrm{nf}} e'$ (i.e. e is an immediate predecessor of e' under out-of-order semantics)
- $e, e' \in R \wedge e <_p e' \wedge snd(e) <_q snd(e')$, for $p, q \in \mathscr{P}$ (i.e. messages cannot overtake each other on the channel, so any two messages sent by a system component to another will arrive in the correct order)

We can now define two semantic functions that assign a Σ-labeled causal poset to a basic sequence diagram.

Definition 3.4. The semantics of a basic sequence diagram $bSD = (E, <_v, <_m, \mathscr{P}, \mathscr{M}, \Sigma, loc, lbl, rcv, snd)$ can be described with a semantic function $[\![\_]\!]^x_{\text{bSD}}$, with $x \in \{\text{nf}, \text{f}\}$, such that

- $[\![bSD]\!]^{\text{nf}}_{\text{bSD}} = (E, \prec_{\text{nf}}, lbl)$, in the presence of out-of-order delivery channels
- $[\![bSD]\!]^{\text{f}}_{\text{bSD}} = (E, \prec_{\text{f}}, lbl)$, in the presence of in-order delivery channels.

In general, when the specific type of channel is not important for the discussion, we will use $[\![\_]\!]_{\text{bSD}}$ as a generic semantic function for basic sequence diagrams.

Basic sequence diagrams can be composed to obtain more complex behaviors. In UML 2 this is possible by means of interaction operators. We consider four operators: `seq` (for weak sequential composition), `alt` (for alternative composition), `par` (for parallel composition) and `loop` (for iterative composition). The weak sequential composition of two sequence diagrams consists in their lifeline-by-lifeline concatenation, such that for each instance, the events of the first diagram precede the events of the second diagram. Events on different lifelines are interleaved. In the parallel composition of two sequence diagrams their events are interleaved. The alternative composition of two sequence diagrams describes a choice between them, such that in any run of the system events will be ordered according to only one of the diagrams. That is, alternative composition introduces alternative orderings of events. The semantics of an alternative composition of basic sequence diagrams is therefore defined by a set of posets. The iterative composition of a sequence diagram can be seen as the weak sequential composition of a number of instances of that sequence diagram.

The syntax of a composite sequence diagram (SD) is defined by the following BNF-grammar:

$$SD \stackrel{def}{=} bSD \mid (SD_1 \,\texttt{seq}\, SD_2) \mid (SD_1 \,\texttt{alt}\, SD_2) \mid (SD_1 \,\texttt{par}\, SD_2) \mid \texttt{loop}(min, max)\, SD_1$$

We describe the semantics of a composite sequence diagram by means of a *set* of causal posets (one for each possible alternative behavior described by the sequence diagram). As we did for basic sequence diagrams, we consider two semantic function $[\![\_]\!]^{\text{nf}}_{\text{SD}}$ and $[\![\_]\!]^{\text{f}}_{\text{SD}}$ that assign a *set* of Σ-labeled posets to a composite sequence diagram.

We introduce now two operations on posets that we need for the definition of $[\![\_]\!]^x_{\text{SD}}$, with $x \in \{\text{nf}, \text{f}\}$. The first one is *weak sequencing* of two Σ-labeled posets (over $\mathscr{P}, \mathscr{M}$), which results in a new Σ-labeled poset where, for each lifeline, the events of the first poset precede the events of the second poset. The second operation is concurrence of two Σ-labeled posets, which results in a new poset with interleaved events.

Definition 3.5 (**Weak sequencing**). Let $p_1 = (E_1, \prec_1, lbl_1)$ and $p_2 = (E_2, \prec_2, lbl_2)$ be two Σ-*labeled* posets (over $\mathscr{P}, \mathscr{M}$) with disjoint sets of events[2]. Their weak sequencing, $p_1 \circ_w p_2$, is a new Σ-*labeled* poset defined as $p_1 \circ_w p_2 = (E_1 \cup E_2, \prec_{1\circ_w 2}, lbl_1 \cup lbl_2)$, where $\prec_{1\circ_w 2} = (\prec_1 \cup \prec_2 \cup \{(e_1, e_2) \in E_1 \times E_2 : loc(e_1) = loc(e_2)\})^*$.

Definition 3.6 (**Concurrence**). Let $p_1 = (E_1, \prec_1, lbl_1)$ and $p_2 = (E_2, \prec_2, lbl_2)$ be two Σ-*labeled* posets (over $\mathscr{P}, \mathscr{M}$) with disjoint sets of events[2]. Their parallel composition, $p_1 \parallel p_2$, is a new Σ-*labeled* poset defined as $p_1 \parallel p_2 = (E_1 \cup E_2, \prec_1 \cup \prec_2, lbl_1 \cup lbl_2)$.

Now we can define the semantics of a composite sequence diagram, along the lines in [KL98], as follows:

Definition 3.7. The semantics of a composite sequence diagram *SD* can be described with a semantic function $[\![\_]\!]^x_{\mathrm{SD}}$, with $x \in \{\mathrm{nf}, \mathrm{f}\}$, such that

$$
\begin{aligned}
[\![bSD]\!]^x_{\mathrm{SD}} &\stackrel{def}{=} \{[\![bSD]\!]^x_{b\mathrm{SD}}\} \\
[\![SD_1 \,\mathtt{seq}\, SD_2]\!]^x_{\mathrm{SD}} &\stackrel{def}{=} \{p_1 \circ_w p_2 : p_1 \in [\![SD_1]\!]^x_{\mathrm{SD}}, p_2 \in [\![SD_2]\!]^x_{\mathrm{SD}}\} \\
[\![SD_1 \,\mathtt{par}\, SD_2]\!]^x_{\mathrm{SD}} &\stackrel{def}{=} \{p_1 \parallel p_2 : p_1 \in [\![SD_1]\!]^x_{\mathrm{SD}}, p_2 \in [\![SD_2]\!]^x_{\mathrm{SD}}\} \\
[\![SD_1 \,\mathtt{alt}\, SD_2]\!]^x_{\mathrm{SD}} &\stackrel{def}{=} [\![SD_1]\!]^x_{\mathrm{SD}} \bigcup [\![SD_2]\!]^x_{\mathrm{SD}} \\
[\![\mathtt{loop}(min, max)\, SD]\!]^x_{\mathrm{SD}} &\stackrel{def}{=} \bigcup_{min \le i \le max} [\![\Delta_i.SD]\!]^x_{\mathrm{SD}}
\end{aligned}
$$

where

$$
\begin{aligned}
\Delta_0.SD &\stackrel{def}{=} \{(\emptyset, \emptyset, \emptyset)\} \\
\Delta_n.SD &\stackrel{def}{=} SD \,\mathtt{seq}\, \Delta_{n-1}.SD, n > 0
\end{aligned}
$$

In general, when the specific type of channel is not important for the discussion, we will use $[\![\_]\!]_{\mathrm{SD}}$ as a generic semantic function for sequence diagrams.

For the analysis of sequence diagrams it is useful to distinguish their initiating and terminating events. For a sequence diagram *SD*, we denote its multi-set of *initiating events* as $init(SD) = \{min(ps) : ps \in [\![SD]\!]_{\mathrm{SD}}\}$, where $min(ps) = \{e \in E : \nexists e' \in E, e' \prec e\}$ is the set of minimum events (i.e. events non-causally dependent on other events) of the Σ-labeled poset *ps*. The initiating events of *SD* will be the minimum sending events for each possible alternative described by the sequence diagram. Similarly, we denote multi-set of *terminating events* for a sequence diagram *SD* as $term(SD) = \{max(ps) : ps \in [\![SD]\!]_{\mathrm{SD}}\}$, where $max(ps) = \{e \in E : \nexists e' \in E, e \prec e'\}$ is the set of maximum events (i.e. events that do not precede any other events)

[2]If they are not disjoint, they are renamed

of the Σ-labeled poset ps. The terminating events of SD will be the maximum receiving events for each possible alternative described by the sequence diagram. Finally, we denote the multi-set of *terminating sending events* of SD as $term_{\text{snd}}(SD) = \{max_{\text{snd}}(ps) : ps \in [\![SD]\!]_{\text{SD}}\}$, where $max_{\text{snd}}(ps) = \{s \in S : \nexists s' \in S, s \prec s'\}$ is the set of maximum sending events (i.e. sending events that do not precede any other sending events, just receiving events) of the Σ-labeled poset ps. The terminating sending events of SD will be the maximum sending events for each possible alternative described by the sequence diagram.

3.2 Syntax and Semantics for Choreography Graphs

We present now the syntax and semantics for choreography graphs.

3.2.1 Choreography Syntax

Each of the activities in the activity diagram of a choreography can be seen as a phase in the execution of a service collaboration C. In each phase or activity, a specific sub-collaboration of C is active (so-called activity's *active collaboration*). This is represented by adorning the activity with a collaboration-use, whose roles are bound to instances of C's roles. For example, in Fig. 4, the *BuyTicket* collaboration is active in the first activity. This is expressed by adorning that activity with a *b:BuyTicket* collaboration-use, whose roles (i.e. P_{bt} and T_{bt}) are bound to instances of *ShuttleService*'s roles (i.e. *P:P* and *T1:T*). The solid circles and bars beside the roles are respectively used to identify the role that initiate and terminate each collaboration. Each activity has one input pin representing the starting execution point of the activity's active collaboration, and one or more output pins representing alternative end-of-execution points of the active collaboration. These pins are represented as small empty rectangles attached to the boundary of the activity node. If several alternative end-of-execution pins exists, each of them is surrounded by an additional rectangle (see Fig. 12 on page 32 for an example). Activities may have additional output pins, describing execution points where the active collaboration is suspended to invoke another collaboration, as well as additional input pins, describing execution points at which a previously suspended active collaboration is to be resumed. Pins used for invoking and resuming an activity's active collaboration are represented as small rectangles with an arrow inside. Both input and output pins represent execution points at which an activity's active collaboration interact with other collaborations. They are labeled with predicates describing goals of the active collaboration.

Edges (i.e. directed connections between activities) and control-flow nodes (i.e. decision, merge, fork, join, initial and final nodes) are respectively used to allow and coordinate the flow of control among activities. An activity can only have one incoming edge, so multiple incoming edges must be AND- or OR-joined.

According to the concrete syntax just described, the formal syntax of goal sequences can be defined as follows:

Definition 3.8 (**Choreography**)**.** A choreography of the sub-collaborations of a collaboration C is a directed graph defined by the tuple $CH = (V, \mathscr{E}, \mathscr{R}_{\text{int}}, \searrow_{\text{ch}}, g_{\text{e}}, m_{\text{exp-a}}, R_{\text{CH}}, AC, m_{\text{a-ac}}, m_{\text{p-a}}, pin, l_{\text{pred}}, p_{\text{type}})$ where

1. V is a set of nodes. It is partitioned into an initial node (v_0) and sub-sets of activities (V_{A}), input pins (V_{InP}), output pins (V_{OutP}), control flow nodes (V_{FLOW}), accept event actions (V_{EA}) and final nodes (V_{FI}). In turn, V_{FLOW} is partitioned into decision (V_{D}), merge (V_{M}), fork (V_{F}) and join (V_{J}) nodes.

2. $\mathscr{E} \subseteq (V_{\text{OutP}} \cup V_{\text{FLOW}} \cup V_{\text{EA}} \cup \{v_0\}) \times (V_{\text{A}} \cup V_{\text{InP}} \cup V_{\text{FI}} \cup V_{\text{EA}} \cup V_{\text{FLOW}})$ is a set of directed edges between nodes, which is partitioned into normal edges ($\mathscr{E}_{\text{n}}$) and interrupting edges ($\mathscr{E}_{\text{int}}$).

3. $\mathscr{R}_{\text{int}}$ is a set of interruptible regions (i.e. regions containing nodes that can be interrupted).

4. $\searrow_{\text{ch}} \subseteq \mathscr{R}_{\text{int}} \times (\mathscr{R}_{\text{int}} \cup V)$ is a hierarchy relation among interruptible regions and nodes. We write $reg \searrow x$ if x is a node or an interruptible region that is directly contained by the interruptible region reg.

5. g_{e} is a guard function for edges. It is defined from $\mathscr{E}_{\text{n}}$ into boolean expressions.

6. $R_{\text{CH}} = \{(id, type) : type \in R_{\text{C}}\}$ is a set of role instances, with R_{C} being the set of roles of collaboration C.

7. AC is a set of *active collaborations*, that is, a collaboration-use representing a specific occurrence of one of C's sub-collaborations. For each $(id, type, B) \in AC$, id is the name of the collaboration-use; $type$ is the name of the collaboration that actually describes the collaboration-use (i.e. one of C's sub-collaborations); and $B \subseteq R_{type} \times R_{\text{CH}}$ is a set of role bindings, where R_{type} is the set of roles of the sub-collaboration named $type$.

8. $m_{\text{a-ac}} : V_{\text{A}} \to AC$ is a non-injective function that maps active collaborations to activities.

9. $m_{\text{p-a}} : V_{\text{InP}} \cup V_{\text{OutP}} \to V_{\text{A}}$ is a function mapping input and output pins to activities, and $pin : V_{\text{A}} \to \mathscr{P}(V_{\text{InP}} \cup V_{\text{OutP}})$ is a function that returns the set of pins attached to a given activity.

10. $l_{\text{pred}} : V_{\text{InP}} \cup V_{\text{OutP}} \to Pre$ is an injective function labeling each input and output pin of an activity with a state predicate of the activity's active collaboration.

11. $p_{\text{type}} : V_{\text{InP}} \cup V_{\text{OutP}} \to \{START, END, INVOCATION, RESUMPTION\}$ is a function that classifies pins as either *starting*, *end-of-execution*, *invocation* or *resumption* ones.

3.2.2 Choreography Semantics

The semantics of a choreography can be intuitively understood in terms of a token-game. At a high level of abstraction, when an activity receives an input token, its active collaboration is enabled. If the token is received on the activity's starting input pin, the active collaboration can begin execution from its initial state. Otherwise, if the token is received through a resuming input pin, the active collaboration can resume execution from the state represented by the event-goal labeling the pin. The active collaboration in reality begins or resumes its execution when one of its roles takes the appropriate initiative. Thereafter, it evolves until an interaction point with other collaborations is eventually reached. That is, the active collaboration runs until the predicate of one of its activity's output pins holds. When this happens, the control token is passed on to the next activity or control node. According to this semantics, the intended behavior of the *ShuttleService* collaboration, as specified by its choreography (see Fig. 4), closely reflects the requirements. Initially the *BuyTicket* collaboration is started and thereafter suspended after the ticket is requested. At that point, a check is performed to determine if the vehicle is at the terminal (i.e. at *T1*). If the result is positive, *BuyTicket* is finished and *EnterVehicle* is enabled, followed by *VehDeparture*, *VehArrival* and *ExitVehicle*. If the vehicle was not at *T1*, this role initiates *ReqVehicle* to request the vehicle from *T2*. *VehDeparture* is then executed, followed by *VehArrival*, which allows *BuyTicket* to be resumed. Thereafter the service progresses as explained before.

The above high-level semantics, which describes the intended behavior of a service collaboration from the point of view of the service designer, was formalized in [CB06b]. This semantics abstract away from individual roles, and implicitly considers that the sequencing between collaborations is strong. That is, when a collaboration passes the control token to the next collaboration through an end-of-execution pin, the behavior of the former collaborations is assumed to be completely finished, for all its participating roles. In the present work we consider a weak sequencing semantics, since it better reflects the actual behavior. Instead of assuming only one control token, we may think that there is one specific token for each role instance appearing in the choreography graph. In order to perform a sending or receiving event, a role needs its token. As soon as a role is finished with its participation on an activity's active collaboration, its token can be sent out to the next activity and the role can start participating in the active collaboration of the new activity. This means that, at any point in time, the execution of two active collaborations may partially overlap. This behavior can be described with a Petri net. However, we will not use Petri nets to formalize the semantics of choreography graphs. Instead we will use partial orders, as we did for sequence diagrams. We note that runs of a Petri net can be described as partial orders over events, where the events correspond to the firing of the net transitions [Kie97].

Paths that start at the initial node of a choreography graph and end at any of the final nodes correspond to finite executions of the service collaboration modeled by the choreography. Infinite paths (due to loops) starting at the choreography's initial node correspond to infinite executions of the service collaboration. A labeled poset can be obtained for each of the finite and infinite execution paths of the

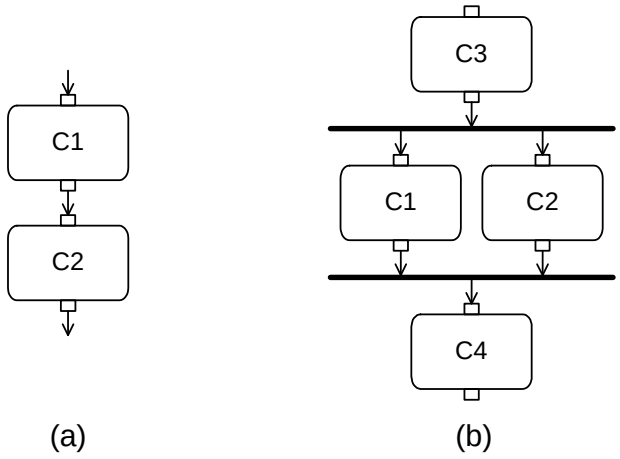

Figure 6: Example of sequential and parallel composition of activities in a choreography graph.

choreography. The semantics of a choreography is then defined as the (possibly infinite) set of labeled posets obtained from all the choreography's finite and infinite executions paths.

The execution paths of a choreography can be obtained using a depth first search technique. The labeled poset corresponding to a given execution path can be obtained by applying some simple guidelines. We detail those guidelines for each of the possible ways in which activities (i.e. collaborations) can be composed or ordered in the choreography graph[3]. In the explanation we consider that the behavior of an activity C_i is described by a collaboration, whose behavior is in turn described by a sequence diagram SD_i.

Sequential composition

When the end-of-execution pin of an activity C_1 is connected (directly, or via one or more control nodes) to the starting pin of an activity C_2, C_1 and C_2 are composed in weak sequence. This is the case for activities C_1 and C_2 in Fig. 6(a), and C_3 and C_1 in Fig. 6(b). When the activities are directly connected (e.g. Fig. 6(a)), or connected through decision and merge nodes, the semantics of the composition corresponds to the `seq` operator defined for sequence diagrams. The semantics for the composition in Fig. 6(a) is thus $[\![SD_1 \,\texttt{seq}\, SD_2]\!]_{\mathrm{SD}}$.

[3]We do not consider here *alternative* composition, which just defines several execution paths. Decision and merge nodes in the choreography graph are used to select one or another path, but they are otherwise ignored in order to build the labeled poset for the selected execution path.

Parallel composition

Activities inside a fork-join pair are composed in parallel (i.e. they are executed concurrently). Since we require proper nesting of fork and join nodes[4], the semantics of such composition corresponds to the `par` operator defined for sequence diagrams. The semantics for the composition of C_1 and C_2 in Fig. 6(b) is thus $[\![SD_1 \,\texttt{par}\, SD_2]\!]_{\mathrm{SD}}$. The semantics for the whole composition presented in Fig. 6(b) is $[\![(SD_3 \,\texttt{seq}\, (SD_1 \,\texttt{par}\, SD_2)) \,\texttt{seq}\, SD_4]\!]_{\mathrm{SD}}$. We note that fork-join pairs are first processed, and then composed with any preceding and/or succeeding activities.

Interruption composition

In this kind of composition an activity C_2 interrupts another activity C_1. It is represented as in Figures 7(a) and 7(b). In Fig. 7(a) an accept event action is enabled whenever C_1 reaches a point in its execution where predicate *pred* holds. From that point of time on, if the event e associated to the accept event action is observed, activity C_1 is interrupted and activity C_2 starts execution. We assume that event e is an external stimulus from the environment observed by the role(s) initiating C_2. If C_1 finishes execution before event e is observed, the interruptible region (i.e. the dashed rectangle) is abandoned via a normal edge. The accept event action is then disabled (i.e. interruption is no longer possible) and activity C_4 is started. In the case of Fig. 7(b), activity C_1 may be interrupted as soon as it starts execution, since the accept event action becomes enabled as soon as the interruptible region is entered.

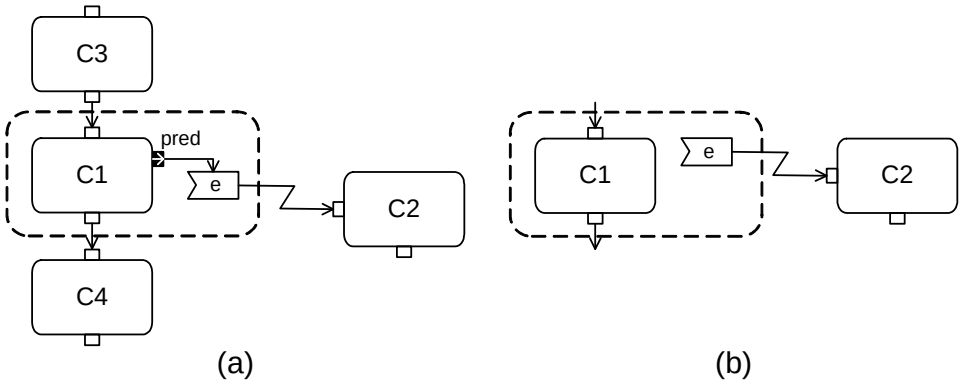

Figure 7: Examples of interrupting composition of activities in a choreography graph.

To define the semantics of interruption we need to introduce the notion of prefix, and prefix with a fixed part, of a labeled poset.

[4]All outgoing edges of a fork node should lead to the same join node, and all incoming edges of a join node should come from the same fork node. An exception is the following. If the join node is connected to a final node, the former could be removed, and let all outgoing edges of the fork lead to final nodes.

Definition 3.9 (**Prefix**). A labeled poset $(E', \prec', lbl')$ is a *prefix* of a labeled poset $(E, \prec, lbl)$ describing the semantics of a sequence diagram SD iff the following conditions hold:

- $E' \subset E$
- if $e \in S'$, then $rcv(e) \in E'$ (i.e. if the prefix contains a sending event, it also contains its corresponding receiving event)
- $\forall e, f$, if $e \prec f \wedge f \in E'$, then $e \in E'$ (i.e. if an event f is part of the prefix, all the events that precede f in the original poset are also in the prefix)
- $\prec' = \prec \cap (E' \times E')$
- $lbl' = lbl \restriction E$, where $\restriction$denotes restriction

Note that the empty poset $[\varepsilon]$ is a prefix of each poset. Note also that this definition differs from the one in [KL98] in two respects. First, we require the set of events of the prefix to be strictly included in the set of events of the original poset. This means that a poset cannot be a prefix of itself. Second, for each sending event contained in the prefix we require its matching receiving event to be also contained in the prefix. For a poset ps we denote the set of all its prefixes as $prefix(ps)$.

Definition 3.10 (**Prefix with fixed part**). A labeled poset $(E', \prec', lbl')$ is a *prefix with a fixed part* of a labeled poset $(E, \prec, lbl)$, with upper limit for the fixed part a predicate event e_{pred} with label *pred (i.e.* $lbl(e_{pred}) = pred$*),* iff the following conditions hold:

- $(E', \prec', lbl')$ is a prefix of $(E, \prec, lbl)$
- $e_{pred} \in E'$ and $\forall f \in E, f \prec e_{pred}$, we have $f \in E'$

A prefix with a fixed part cannot be an empty poset. It will always contain a fixed part consisting of the predicate event e_{pred} and all its predecessor events in the original poset. For a poset ps and a predicate $pred$ labeling one of ps's events, we denote the set of all its prefixes with a fixed part as $fprefix(ps, pred)$.

We let $C_2 \,\texttt{int}\, C_1$ informally mean that activity C_2 interrupts activity C_1 from the beginning of C_1 (as in Fig. 7(b)). The semantics for this type of interrupting composition is defined as follows:

$$[\![C_2 \,\texttt{int}\, C_1]\!] = \{ps_1 \circ_w ps_2 : ps' \in [\![SD_1]\!]_{\mathrm{SD}}, ps_1 \in prefix(ps'), ps_2 \in [\![SD_2]\!]_{\mathrm{SD}}\}$$

Valid posets for the interruption are those formed by the weak sequencing of a prefix of one of C_1's posets and one of C_2's posets.

We now let $C_2 \,\texttt{int}(pred)\, C_1$ informally mean that activity C_2 interrupts activity C_1 from the point in the execution of C_1 where predicate *pred* holds (as in Fig. 7(a)). We also let e_{pred} be the predicate event with label *pred* (i.e. $lbl(e_{pred}) = pred$). The semantics for this type of interrupting composition can then be defined as follows:

$$[\![C_2 \,\texttt{int}(pred)\, C_1]\!] = \{ps_1 \circ_w ps_2 : ps' = (E, \prec, lbl) \in [\![SD_1]\!]_{\mathrm{SD}} \text{ such that } e_{pred} \in E,$$
$$ps_1 \in fprefix(ps', pred), ps_2 \in [\![SD_2]\!]_{\mathrm{SD}}\}$$

Since SD_1 may have several associated posets (in case it contains alternatives or loops), we select those ps' posets that contain e_{pred}. Valid posets for the interruption are those formed by the weak sequencing of a prefix with a fixed part of ps' and one of C_2's posets. The fixed part is the behavior the needs to be executed in order for predicate *pred* to hold.

We note that the above semantics describe cases where the interruption actually happens, that is, where an interrupting edge is traversed. The semantics for the choreography in Fig. 7(a) would be

$$[\![(SD_3 \,\texttt{seq}\, SD_1)\,\texttt{seq}\, SD_4]\!]_{\mathrm{SD}} \cup [\![SD_3 \,\texttt{seq}\, (C_2\,\texttt{int}(pred)\, C_1)]\!]$$

Invocation composition

In the most general case, this type of composition implies that while in the middle of its execution, C_1 invokes C_2. Thereafter, the behaviors of C_1 and C_2 may proceed independently. Here we consider a more specific case of invocation (so-called *invocation with feedback*), where C_1, after reaching a point in its execution where a predicate *pred* holds, invokes C_2. C_1 then is suspended and waits for C_2 to execute all or part of its behavior before resuming. This normally represent a goal dependency between C_1 and C_2, such that C_1 can only achieve its own goal if C_2 achieves its corresponding one. An example is shown in Fig. 8(a).

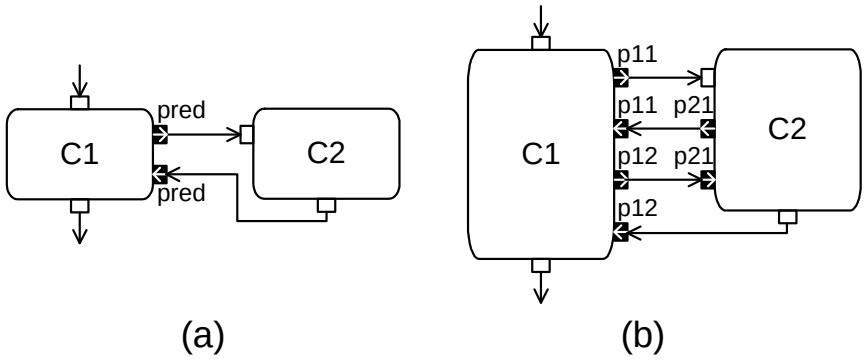

Figure 8: Examples of invocation composition of activities in a choreography graph.

Invocation composition is achieved with help of invocation and resumption pins. For each invocation output pin labeled with a predicate *pred*, there should be a corresponding resumption input pin labeled with the same predicate (see Fig. 8). These pins correspond to execution points that are represented by means of predicate events in the sequence diagram of the invoking activity. For each pair of invocation/resumption pins of an invoking activity that are labeled with a predicate *pred*, the sequence diagram describing the behavior of that activity should contain a predicate event labeled with the same predicate *pred*. Since the invoking activity is required to get suspended after the invocation, such predicate event may not appear inside an operand of a `par` combined fragment.

To define the semantics of invocation we need to introduce the notion of segment of a labeled poset.

Definition 3.11 (**Segment**). A labeled poset $(E', \prec', lbl')$ is a *segment* of a labeled poset $ps = (E, \prec, lbl)$ with predicates $pred_1$ and $pred_2$ as lower and upper limits (written $seg(ps, pred_1, pred_2)$), iff the following conditions hold:

- Let $e_1, e_2 \in E$ such that $lbl(e_1) = pred_1$ and $lbl(e_2) = pred_2$, and let $E_{\text{pr}} = \{f \in E : f \prec e_2\}$ and $E_{\text{sc}} = \{f \in E : e_1 \prec f\}$,
  - if $pred_1 = start$, then $E' = \{e_2\} \cup E_{\text{pr}}$ (i.e. the segment contains e_2 and all events of the original poset that are predecessors of e_2)
  - if $pred_2 = end$, then $E' = E_{\text{sc}}$ (i.e. the segment contains all events of the original poset that are successors of e_1)
  - if $pred_1 \neq start \wedge pred_2 \neq end$, then $E' = \{e_2\} \cup (E_{\text{sc}} \cap E_{\text{pr}})$ (i.e. the segment contains e_2 and all events of the original poset that are successors of e_1 and predecessors of e_2)
  - if $pred_1 = start \wedge pred_2 = end$, then $E' = E$ (i.e. the segment is the original poset)
- if $e \in S'$, then $rcv(e) \in E'$ (i.e. if the segment contains a sending event, it also contains its corresponding receiving event)
- $\prec' = \prec \cap (E' \times E')$
- $lbl' = lbl \restriction E$, where $\restriction$denotes restriction

In the above definition, *start* and *end* are special purpose predicates that should not label any of the predicate events of the original poset.

We let C_1 `inv`(Ψ_1, Ψ_2) C_2 informally mean that activities C_1 and C_2 invoke each other (with C_1 performing the first invocation) at the execution points indicated by the predicates in Ψ_1 and Ψ_2. Ψ_1 (resp. Ψ_2) is an ordered set containing the predicates that hold at execution points where C_1 (resp. C_2) invokes C_2 (resp. C_1). We now explain how to obtain Ψ_1 for each $ps'_1 \in [\![SD_1]\!]_{\text{SD}}$ (the same procedure can be used to obtain Ψ_2):

1. Get the predicates labeling the invocation pins of C_1 that are connected to pins of C_2, that is, $Pr = \{l_{\text{pred}}(ip) : ip \in pin(C_1), p_{\text{type}}(ip) = INVOCATION, (ip, x) \in \mathscr{E}, m_{\text{p-a}}(x) = C_2\}$
2. Get the total ordered set t_{pred} of predicate events in $ps'_1 = (E'_1, \prec'_1, lbl'_1)$ that are labeled with predicates from Pr (i.e. events $e \in E'_1$ such that $lbl'_1(e) \in Pr$, and their order relations). Note that it is a total order since the predicate events denoting invocation points cannot appear inside `par` combined fragments of C_1's sequence diagram to enforce suspension.
3. Replacing each event in t_{pred} with its label, and adding the special purpose predicates *start* and *end* as first and last elements, respectively, we obtain Ψ_1

For the invocation composition in Fig. 8(a) we would have $\Psi_1 = \{start, pred, end\}$ and $\Psi_2 = \{start, end\}$, while for the invocation composition in Fig. 8(b) we would have $\Psi_1 = \{start, p_{11}, p_{12}, end\}$ and $\Psi_2 = \{start, p_{21}, end\}$.

In general, we assume that $\Psi_1 = \{pred_1^1, \ldots, pred_1^n\}$ (with $pred_1^1 = start$ and $pred_1^n = end$) and $\Psi_2 = \{pred_2^1, \ldots, pred_2^{n-1}\}$ (with $pred_2^1 = start$ and $pred_2^{n-1} = end$), where $n > 1$. The semantics for $C_1 \,\texttt{inv}(\Psi_1, \Psi_2)\, C_2$ can then be defined as follows:

$$\begin{aligned}[\![C_1 \,\texttt{inv}(\Psi_1, \Psi_2)\, C_2]\!] = \{&seg(ps_1', pred_1^1, pred_1^2) \circ_w seg(ps_2', pred_2^1, pred_2^2) \circ_w \ldots \\ &\ldots seg(ps_2', pred_2^{n-2}, pred_2^{n-1}) \circ_w seg(ps_1', pred_1^{n-1}, pred_1^n) : \\ &ps_1' \in [\![SD_1]\!]_{\mathrm{SD}}, ps_2' \in [\![SD_2]\!]_{\mathrm{SD}}\}\end{aligned}$$

4 Realizability of Choreographies

In the following sections we study the direct realizability of a choreography from the point of view of the operators used to compose the sub-collaborations. In our discussion we assume that each sub-collaboration referred to in a choreography is directly realizable. Starting from single messages and applying the operators and rules described in the following will ensure this. For the sequential, alternative, parallel and interruption composition operators we study the problems that can lead to difficulties of realization. We investigate the actual nature of these problems and discuss possible solutions to prevent or remedy them.

4.1 Sequential Composition

Sequential composition imposes a causal dependency or partial order between the events of the composed sub-collaborations. In the following the notions of strong and weak sequential composition are discussed.

Strong Sequencing

Strong sequencing between two collaborations C_1 and C_2, written $C_1 \circ_s C_2$, requires C_1 to be completely finished, for all its components, before C_2 can be initiated. It requires a direct precedence relation between the terminating action(s) of C_1 and the initiating action(s) of C_2, so that the latter can only happen after the former are finished. This leads to the following:

Proposition 4.1. *The strong sequential composition of two directly realizable collaborations C_1 and C_2, $C_1 \circ_s C_2$, is directly realizable if all terminating actions of C_1 and all initiating actions of C_2 are located at the same component.*

The above proposition requires C_1 to terminate at the component initiating C_2. This is the only way the initiator of C_2 can know when C_1 is completely finished. If this condition is not satisfied, coordination messages must be added from C_1's terminating components to C_2's initiating components, in order to guarantee the strong sequencing. This could be done automatically by a synthesis algorithm [BG86].

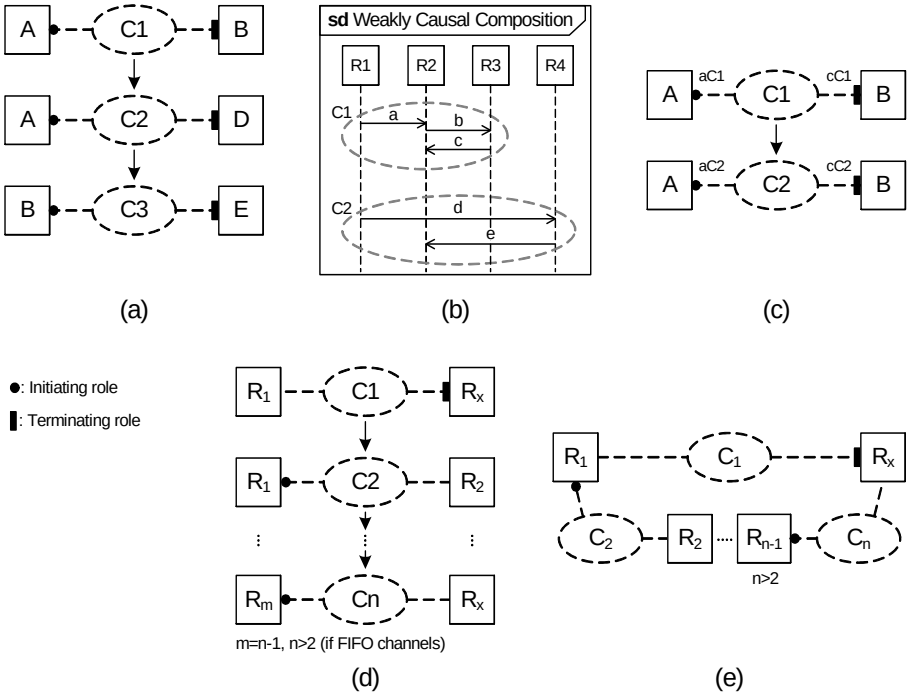

Figure 9: Weak Sequence Problems

Weak Sequencing

Weak sequencing of two sub-collaborations C_1 and C_2, written $C_1 \circ_w C_2$, does not require C_1 to be completely finished before C_2 can be initiated. Any component can start participating in C_2 as soon as it has finished with C_1 (without waiting for the other components to finish as well), which means that the actions from both collaborations are sequenced on a per-component basis. This is the sequential composition semantics used in HMSCs and UML Interaction Overview Diagrams, but not in standard UML activity diagrams. The semantics of choreography graphs presented in the previous section assumes weak sequencing.

Weak sequencing introduces a certain degree of concurrency, since the executions of the composed collaborations may partially overlap. Although such concurrency may be desirable for performance or timing reasons (i.e. a component may initiate a new collaboration if the actions in that collaboration were independent of the actions that have yet to be executed in the first collaboration), it comes at a price, since it may lead to specifications that are not directly realizable and even counter-intuitive. The specification in Fig. 9(a) is an example of a counter-intuitive composition. According to the weak sequence semantics, component B may initiate collaboration C_3 as soon as it has finished with C_1. As a result, collaborations C_2 and C_3 may be executed in any order in the realized system. This is counter-intuitive to the specification, which we assume reflects the designer's intention (i.e. that C_3 should be executed after C_2, with some allowed overlapping). If the designer's intention was that the collaborations should be concurrently executed, this should rather be explicitly specified by means of parallel composition.

To avoid the aforementioned problem, when two collaborations are composed in weak sequence the component initiating the second collaboration should participate in the first collaboration (e.g. as in the composition of C_1 and C_2 in Fig. 9(a)). We say a sequential composition with this property is weakly-causal:

Definition 4.2 (**weak-causality**)**.** The weak sequential composition of two collaborations, $C_1 \circ_w C_2$, is *weakly-causal* if the initiator of C_2 participates in C_1.

A weak sequential composition without the weak-causality property (e.g. the composition of C2 and C3 in Fig. 9(a)) can be made weakly-causal by means of a synchronization message sent from one of the participants of the first collaboration to the initiator of the second collaboration[5].

Weak-causality is a necessary condition for direct realizability of weak sequential composition. However, it is not strong enough to be a sufficient condition. For example, consider the weak sequential composition of C_1 and C_2 in Fig. 9(b). This composition is weakly-causal, but it is not directly realizable. Component *R1* may initiate collaboration C_2 just after sending message a in C_1. Therefore, the actions in C_1 that follow the sending of message a may overlap with those performed in C_2 by the same components. For example, message e may be received at *R2* before message c, or even before message a. Obviously, this message reception order has not been explicitly specified. We note that weak-causality is enforced in the so-called local-HMSCs of [GMSZ06].

In the literature about MSCs, the possibility that messages may be received in a different order from the one specified is usually called a **race condition** [AHP96]. In general, race conditions can occur when a receiving event is specified to happen before another event (i.e. either a receiving or a sending one), and both events are located on the same component. The reason lies in the controllability of events. While a component can always control when its sending events should happen (e.g. it can wait for one or more messages to be received before sending a message), it cannot control the timing of its receiving events. The occurrence of races highly depends on the underlying communication service that is used. If no assumption is made about the communication service, races can only be prevented if all message transmissions are strongly sequenced. This condition might be quite restrictive. We now present a less restrictive condition that does not prevent all races, but reduces their number and facilitates their detection, compared with weak-causality. This condition, which we call *send-causality*, requires all sending events to be ordered, except those that have been explicitly specified (with parallel composition) to happen concurrently.

Definition 4.3 (**send-causal composition**)**.** $C_1 \circ_w C_2$ is *send-causal* if 1) C_1 and C_2 are send-causal, and 2) the component initiating C_2 is the one that performs either the last sending event of C_1 or the receiving event corresponding to that sending event.

[5]The component sending this message should be chosen among the components that participate in both collaborations (if any), in order to minimize the risk of introducing race conditions.

An elementary collaboration is send-causal if can be decomposed into a choreography of sub-collaborations, each of them consisting of exactly one message, where all sequential compositions in the choreography are send-causal. We can give a more formal definition based on sequence diagrams as follows:

Definition 4.4 (**send-causal elementary collaboration**)**.** An elementary collaboration is *send-causal* if its associated sequence diagram is send-causal.

Definition 4.5 (**send-causal sequence diagram**)**.** A sequence diagram is *send-causal* if any of the following conditions is satisfied:

(i) If the diagram is a basic sequence diagram, the following holds: $\forall s, s' \in S$, if $s <_m s' \wedge \nexists s'' \in S, s <_m s'' <_m s'$ then $loc(s') = loc(s) \vee loc(s') = loc(rcv(s))$.

(ii) If the diagram is a composite sequence diagram, the following holds:

- All its basic sequence sub-diagrams are send-causal
- Whenever two sub-diagrams SD_1 and SD_2 are composed in weak sequence (i.e. $SD_1 \,\mathtt{seq}\, SD_2$), the following is satisfied: $\forall \mathscr{T}_S \in term_{\text{snd}}(SD_1), \forall \mathscr{I} \in init(SD_2), \forall s_t \in \mathscr{T}_S, \forall s_i \in \mathscr{I}, loc(s_i) = loc(s_t) \vee loc(s_i) = loc(rcv(s_t))$.

Note that in condition (ii) of Definition 4.5 we have implicitly considered the possibility that a composite sequence diagrams may describe alternative or parallel behaviors. In such situation, for each alternative behavior, or each parallel behavior, we require that the send-causality property holds.

It can be shown that when send-causality is enforced, races may only occur between two or more consecutive receiving events (i.e. not between a sending event and a receiving event).

Proposition 4.6. *In a send-causal composition race conditions may only exist between two or more consecutive receiving events.*

Proof. See Appendix A. □

Corollary 4.7. *A send-causal composition is directly realizable over a communication service with in-order delivery and separate input buffers.*

One of our motivations is to provide guidelines for constructing specifications with as few conflicts as possible and whose intuitive interpretation corresponds to the behavior allowed by the underlying semantics. We therefore propose, as a general specification guideline, that all elementary collaborations be send-causal. Weak sequencing of collaborations should also be send-causal, unless there is a good reason to relax this requirement. In the following we assume that all elementary collaborations are send-causal.

A *potential* race condition exists between two weakly sequenced collaborations, $C_1 \circ_w C_2$, if there is a component that participates in both collaborations playing roles

that may partially overlap. Due to Proposition 4.6, if the sequencing is send-causal this may only happen when the role that the component plays in C_1 ends with a message reception (i.e. it is a terminating role) and the role it plays in C_2 starts with another message reception (i.e. it is a non-initiating role). Whether a potential race condition is an *actual* race or not depends on the underlying communication service, and on whether messages are received from the same or from different components. For example, in Fig. 9(c) a potential race condition exists at component B between the receptions of the last message in C_1 and the first message in C_2, but it is only actual in the case of out-of-order delivery.

We note that race conditions may not only appear between *directly* composed collaborations (Fig. 9(c)), but also between *indirectly* composed ones, as shown in Fig. 9(d). In this specification it is the weak sequencing between C_1 and C_2 that makes the potential race between C_1 and C_n possible. We therefore say that there is **indirect weak sequencing** between C_1 and C_n. This "propagation" of weak sequencing makes it more difficult to avoid races.

We have the following result:

Proposition 4.8. *The* send-causal *weak sequential composition of a set of directly-realizable collaborations is directly realizable*

- *over a communication service with* in-order delivery *if the following condition is satisfied: if a component plays a terminating role in a collaboration C_1 followed by a non-initiating role in another collaboration C_n, then the last message it receives in C_1 and the first one it receives in C_n are sent by the same peer-components.*

- *over a communication service with* out-of-order delivery *only if no component plays a terminating role followed by a non-initiating role.*

Working with binary collaborations we can easily know which component sends the first and last messages of a collaboration, if we know which components play the initiating and terminating roles. Due to Proposition 4.8, actual races can then be detected at an early specification stage, when the detailed behavior of each collaboration has not yet been specified, but only the selection of their initiating and terminating roles has been done. In the case of n-ary collaborations, we can perform the same early analysis, but only potential races can be discovered.

One interesting thing of the specification with collaborations is that we can get information about potential races from the diagram describing the structural composition of collaborations (see e.g. Fig. 9(d)). In such diagram we can see whether a component participates in several collaborations, and whether it plays at least one terminating and one non-initiating role in them. If that is the case, a potential race exists. This information could then be used to direct the analysis of the behavioral specification (i.e. the choreography).

Resolution of Race Conditions. Race conditions can be resolved in several ways. Some authors [Mit05, CKS05] have proposed mechanisms to automatically eliminate race conditions by means of synchronization messages. We note that when

the send-causality property is satisfied, the synchronization message should be used to transform the weak sequencing leading to the race into strong sequencing. If synchronization messages are added in other places new races may be introduced.

Other authors tackle the resolution of race conditions at the design and implementation levels. They differentiate between the reception and consumption of messages. This distinction allows messages to be consumed in an order determined by the receiving component, independently of their arrival order. We call this *message reordering for consumption*. In general, this reordering may be implemented by first keeping all received messages in a (unordered) pool of messages. When the behavior of the component expects the reception of one or a set of alternative messages, it waits until one of these messages is available in the message pool. Khendek et al. [KZ05] use the SDL Save construct to specify such message reordering. This technique can be used to resolve races between messages received from the same source (i.e. in the case of channels with out-of-order delivery), as well as races between messages received from different sources. In the latter case, a communication service with separate input buffers would also resolve the races. Finally, races may also be eliminated if an explicit consumption of messages in all possible orders is implemented (i.e. similar to co-regions in MSCs).

We believe that the resolution of races heavily depends on the specific application domain and requirements, as well as in the context which they happen in. In some cases the addition of synchronization messages is not an option, and a race has to be resolved by reordering for consumption. In other cases, such as when races lead to race propagation problems (see Section 4.2) a strict order between receptions is required, so components should be synchronized by extra messages. At any rate, all race conditions should be brought to the attention of the designer once discovered. She could then decide, first, whether the detected race entails a real problem (e.g. in Fig. 9(d) there is no race if all channels have the same latency). Then, she could decide whether reordering for consumption is acceptable or synchronization messages need to be added or the specification has to be revised.

Loops

Loops can be used to describe the repeated execution of a (composite) collaboration, which we call the body collaboration. A loop can therefore be seen as a shortcut for strong or weak sequential composition of several executions of the same body collaboration. This means that the rules for strong/weak sequencing must be applied. We note that all executions of a loop involve the same set of components (weak-causality property is thus always satisfied). This fact makes the chances for races high when weak sequencing is used. Strong sequencing should therefore be preferred for loop bodies in the general case.

Loops may give rise to so-called *process divergence* [BAL97], characterized by a component sending an unbounded number of messages ahead of the receiving component. This may happen if the communication between any two of the participants in the body collaboration is unidirectional (i.e. only happens in one direction).

As we will see in the next section, loops may also affect the realizability of choices.

4.2 Alternative Composition

Alternative composition is specified by means of choice operators, and describes alternatives between different execution paths. In a choice one or more *choosing* components decide the alternative of the choice to be executed, based on the (implicit or explicit) conditions associated with the alternatives. The other *non-choosing* components involved in the choice follow the decision made by the choosing components (i.e. execute the alternative chosen by the later ones). It is therefore important that:

1. The choosing components, if several, agree on the alternative to be executed. We call this the **decision-making process**.

2. The decision made by the choosing components is correctly propagated to the non-choosing components. We call this the **choice-propagation process**.

In the following we study how each of these aspects affect the realizability of a choice. We assume that the set of choosing components is the union of the initiating components of all the choice alternatives.

Decision-making Process

The intuitive interpretation of a choice is that only one of the alternative behaviors is to be eventually executed. Deciding which alternative to be executed becomes simple if there is only one choosing component, and the conditions for the alternatives are local to that component (i.e. they are expressed in terms of observable predicates). Choices with this property are called **local**. It is easy to see that local choices are realizable (up to the decision-making process), since the decision is made by a single component based only on its local knowledge.

The decision-making process gets complicated when there is more than one choosing component. This is the case in the choice of Fig. 10(a), where there are two choosing components, namely A and B. From a global perspective, we may think that once the decision node is reached, either component A initiates collaboration *disc1* with B, or component B initiates collaboration *disc2* with A. We are assuming then that there is an implicit synchronization between A and B, which allows them to agree on the alternative to be executed. However, in a directly realized system, components A and B will not be able to synchronize and they may decide to initiate both collaborations simultaneously.

Choices involving more than one choosing component are usually called **non-local choices** [BAL97]. They are normally considered as pathologies that can lead to misunderstanding and unspecified behaviors, and algorithms have been proposed to detect them in the context of HMSCs (e.g. [BAL97, Hél01]). Despite the extensive attention they have received, there is no consensus on how they should be treated. We believe this is due to a lack of understanding of their nature. Some authors (e.g. [BAL97]) consider them as the result of an underspecification and suggest their elimination. This is done by introducing explicit coordination, as a refinement step towards the design. Other authors look at non-local choices as an obstacle for realizability and propose a restricted version of HMSCs, called *local*

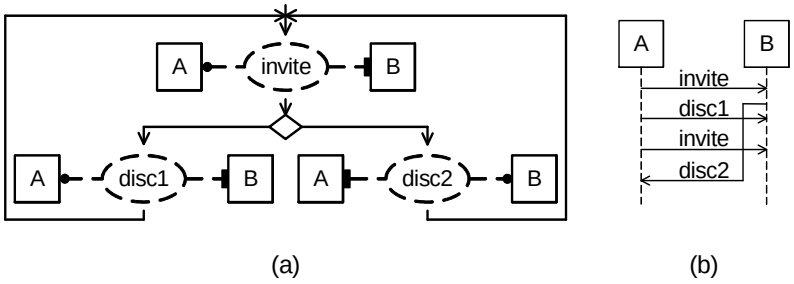

Figure 10: Competing-initiatives choice

HMSCs [HJ00, GMSZ06], that are always realizable. These HMSCs forbid non-local choices. Finally, there are authors [GY84, MGR05] that consider non-local choices to be almost inevitable in the specification of distributed systems with autonomous processes. They propose to address them at the implementation level, and propose a generic implementation approach of non-local choices.

The problem with non-local choices is the existence of several *uncoordinated* components that have the possibility to make an independent decision in the directly realized system. As a solution, we may think of making the choice local by coordinating these components (i.e. either with additional messages or with additional message contents), so that they make a common decision. Such coordination may however not be feasible in all contexts and application domains. Consider, for example, the specification of a personal communication service where both end-users can take the initiative to disconnect. This could be specified as a non-local choice between two disconnection collaborations, each of them initiated by a different component (see Fig. 10(a)). The decision made by any of the components to initiate one of the disconnection collaborations is not totally controlled by that component, but it is triggered by the respective end-user. It makes therefore little sense to coordinate the components in order to obtain a local choice, since this would imply the coordination of the end-users' initiatives. Such non-local choice is simply unavoidable.

We refer to non-local choices where the coordination of the choosing components is not feasible as **competing-initiatives choices**. A characteristic of them is that all the alternative collaborations are simultaneously enabled, and will be triggered by events that cannot be controlled by the initiating components, such as an end-user initiative or a time-out. As a result, the alternative collaborations cannot be prevented from being simultaneously triggered. If this happens, it should be detected as soon as possible and resolved by means of a proper conflict resolution. Any component involved in two or more alternatives may be potentially used to detect the initiative conflict and initiate the resolution. For such components, the competing initiatives reveal themselves in the components' role sequences as choices between an initiating and a non-initiating role, or between two non-initiating roles played in collaborations with different peers.

A side effect of competing-initiatives choices is the existence of **orphan** messages. Consider again the specification in Fig. 10(a), which describes the repetitive

execution of collaboration *invite* followed by either *disc1* or *disc2*. Now imagine that each collaboration consists only of one message. Then the scenario in Fig. 10(b) is possible, where message *disc2* is sent as a response to the first *invite* message, but it is received by A after having sent the second *invite*. Component A may then consume message *disc2* as a response to the second *invite* message, leading to an undesired behavior. In this scenario the collaboration occurrence where *disc2* is sent is considered finished while *disc2* is still in the system (i.e. not consumed). This message becomes thus orphan, with the danger of being consumed in a latter occurrence of the same collaboration. To avoid this messages should be marked (e.g. with a session id), so they are only consumed within the right collaboration instance.

Competing-initiatives choices correspond to the non-local choices discussed by Gouda et al. [GY84] and Mooij et al. [MGR05]. These authors propose some resolution approaches. In the domain of communication protocols, Gouda et al. [GY84] proposes a resolution approach for two competing alternatives (i.e. two choosing components), which gives different priorities to the alternatives. Once a conflict is detected, the alternative with lowest priority is abandoned. With motivation from a different domain, where Gouda's approach is not satisfactory, Mooij et al. [MGR05] propose a resolution technique that executes the alternatives in sequential order (according to their priorities), and is valid for more than two choosing components. We conclude that the resolution approach to be implemented depends on the specific application domain. We therefore envision a catalog of domain specific resolution patterns from which a designer may choose the one that better fits the necessities of her system. We note that any potential resolution should also address the problem of orphan messages, which is not considered in either [GY84] or [MGR05].

Choice-propagation Process

The fact that a choice is local does not guarantee its realizability. The decision made by the choosing component must be properly propagated to the non-choosing components, in order for them to execute the right alternative. In each alternative, the behavior of a non-choosing component begins with the reception of a sequence of messages, which we call the *triggering trace*. Thereafter, the component may send and receive other messages. It the triggering traces that enable a non-choosing component to determine the alternative chosen by the choosing component. In some cases, however, a non-choosing component may not be able to determine the decision made by the choosing component. As an example, we consider the local choice in Fig. 11(a). For the component *R3*, the triggering traces for both alternatives are the same (i.e. the reception of message x). Therefore, upon reception of x, *R3* cannot determine whether *R1* decided to execute collaboration C_1 or C_2. That is, *R1*'s decision is ambiguously propagated to *R3*. We say a choice has an **ambiguous propagation**if there is a non-choosing component for which the triggering traces *specified* in two alternatives have a common a prefix[6]. Choices with ambiguous

[6]Note that this definition considers that there is ambiguous propagation with the following triggering traces: $\{?x, ?y\}$ and $\{?x, ?z\}$. This is true in any directly realized system, since the choice cannot be made immediately after $?x$. An easy solution in this case would be to delay the choice

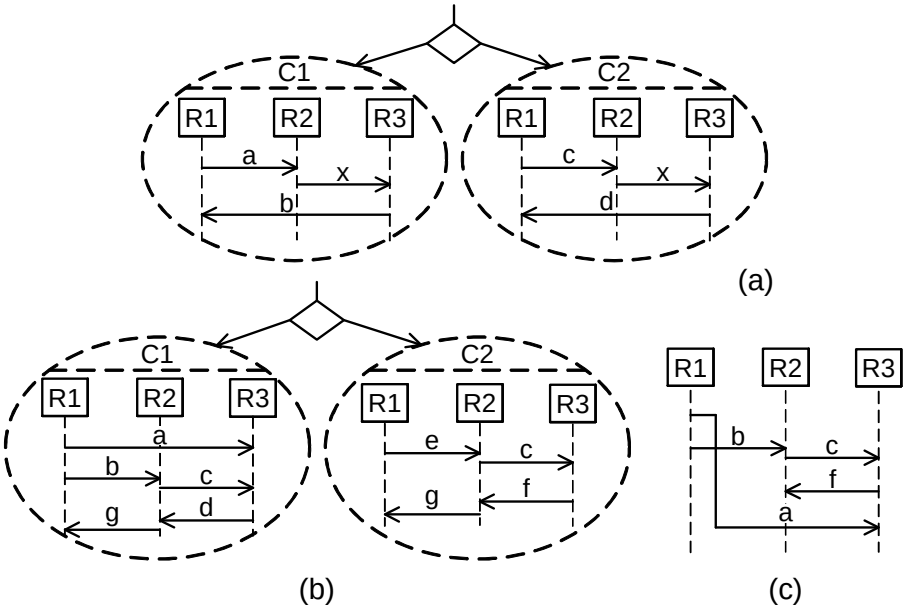

Figure 11: (a) Non-deterministic and (b) Race choice propagation; (c) Behavior implied by (b)

propagation are not directly realizable. They are similar to the non-deterministic choices defined in [MRW06].

Now consider the choice in Fig. 11(b). It is a local choice and, according to the triggering traces specified for any of the two non-choosing components, the propagation should not be ambiguous. Still, this choice is not directly realizable. A race condition between messages a and c in $\mathcal{C}_1$ may lead to the scenario of Fig. 11(c), where *R1* and *R2* execute $\mathcal{C}_1$, while *R3* executes $\mathcal{C}_2$. This example shows that in the presence of race conditions the triggering trace *observed* by a non-choosing component may differ from the specified one. Therefore, whenever race conditions may appear in any of the alternatives, we need to consider the potentially observable triggering traces in the analysis of choice propagation (e.g. {?a, ?c} and {?c, ?a} for *R3* in collaboration $\mathcal{C}_1$ – Fig. 11(b)). We say a choice has a **race propagation** if there is ambiguous propagation due to races. Choices with race propagation are not directly realizable. They are similar to the race choices defined in [MRW06].

To resolve the problem of race propagation we need to eliminate the race(s) that lead to it. However, if we try to remove the race conditions by means of message reordering for consumption (e.g. by means of separate input buffers), the race propagation problem may still persist. This is because, in general, a component would not be able to determine whether a received message should be immediately consumed as part of one alternative, or be kept for later consumption in another alternative (e.g. race propagation in Fig. 11(b) cannot be solved with separate input buffers). To make the message reordering work, we need to mark the messages with the col-

(i.e. extract ?x from the choice). Note, however, that this solution would not always be appropriate (e.g. with the following triggering traces: {?x} and {?x, ?z}).

laboration instance[7] they belong to [BG86]. This not only avoids race propagation, but also ambiguous propagation in general. In [GMSZ06], although choice propagation is not explicitly discussed, the authors propose marking all messages (i.e. not only those involved in a race propagation) as just explained, in order to realize local-HMSCs specifications. Components then have to check the data carried by messages upon each reception. We believe this unnecessarily increases the amount of processing that each component has to do upon message reception. We would prefer to detect the cases of race propagation and either remove the race condition(s) by transforming the responsible weak sequencing into strong sequencing, or apply message reordering together with marking only to the messages involved.

Neither ambiguous nor race choice propagation can be detected at the collaboration level[8], we need to consider the detailed behavior of the sub-collaborations involved in the choice.

A choice without ambiguous or race propagation is said to have **proper decision propagation**. A local choice with proper decision propagation is directly realizable.

4.3 Interruption

The interruption semantics requires a collaboration C be interrupted once another preempting collaboration C_{int} is initiated. In a distributed asynchronous system the interruption may take some time to propagate to all participants in the interrupted collaboration. This means that certain components may still proceed executing their behavior in C for some time after C_{int} has been initiated. For example, a client may send a request to a server and, shortly after that, decide to send a cancellation message. While this message is on the way, the server would continue processing the request, and may even send a response back to the client before it receives the cancellation message. The client would then receive a response message that it does not expect. Similarly, the server would receive a non-awaited cancellation message.

As competing-initiatives choices, interruption compositions suffer from a problem of initiatives (from the interrupted and the interrupting collaborations) that compete with each other. They are therefore not directly realizable, in the general case. Note, however, that the presence of competing initiatives is visible with interruptions, and so the detection is easy at the choreography level. We refer to Section 4.2 for a discussion on resolution of competing initiatives situations and related problems.

4.4 Parallel Composition

A parallel composition is directly realizable as long as the composed collaborations are completely independent (i.e. their executions do not interfere with each other). Unfortunately, sometimes there are implicit dependencies that may lead to unspecified behaviors. This is the case if a component participates in several concurrent

[7] If the choice is part of the body of a loop, the iteration number should be considered

[8] In the case of race propagation we may detect the existence of a race at the collaboration level, but could not determine if that race affects the propagation.

collaborations that use the same message types. Messages belonging to one collaboration may then be consumed within a different collaboration.

Implicit dependencies may also exist through shared resources. In this case, appropriate coordination has to be added between the collaborations, which will normally be service-specific. In [CB06b] we discussed the automatic detection of interactions, due to shared resources, between concurrent instances of the same composite service collaboration. This detection approach makes use of pre- and post-conditions associated with sub-collaborations, and could also be used to detect interactions between collaborations composed in parallel with forks.

Forks, Joins and Sequential Composition.

We note that the rules for (weak/strong) sequential composition should be applied, as in any other context, in the presence of forks and joins. If strong sequencing is required, all the collaborations immediately following a fork should be initiated by the component terminating the collaboration preceeding the fork. Similarly, all the collaborations immediately preceding a join should terminate at the component initiating the collaboration following the join. If weak sequencing is required, each collaboration immediately following a fork should be initiated by a component participating in the collaboration preceeding the fork. Likewise, the component initiating the collaboration following a join should participate in each of the collaborations immediately preceding the join.

We also note that if synchronization behavior is needed in order to guarantee the strong or weak sequencing as explained above (or to remove race conditions), such behavior should be added to the affected branch after the fork, or before the join, in order to prevent interactions with the collaborations in the other branches.

4.5 Conflicts between Concurrent Collaboration Instances

So far we have discussed the conflicts that may appear when sub-collaborations are composed within the scope of an enclosing collaboration. In a running system there will normally be many collaborations executing in parallel. One will normally not define the complete system behavior explicitly as one collaboration, but rather let it be implied from the binding (and composition) of roles to components. Here we briefly discuss the problem of undesired interactions between concurrent instances of the same composite collaboration (e.g. concurrent sessions of a service).

If several instances of a collaboration are executed at the same time, and they involve disjoint sets of components (i.e. the roles of each collaboration instance are bound to different components), they will run independently, without interactions. The situation is however different if one component participates in two or more collaboration instances. In that case, undesired interactions between the collaborations may arise if the roles played by the component in those collaborations need to access shared resources. To avoid such interactions the roles should be properly

coordinated. Depending on the kind of resource and on the concrete service requirements, a different mechanism may be needed for their coordination. One may also distinguish between static role binding, which is resolved at design time, and dynamic role binding, which is resolved at runtime. For example, an FTP server may maintain concurrent sessions with different clients. Since the memory at the server is a shared and limited resource, the total number of such concurrent sessions should be restricted to a maximum. A session manager could then be used that would dynamically bind new session roles or reject session requests once the maximum was reached. In a telephone service, where a user may receive a call while already talking to someone, a call-waiting functionality may be a better solution.

In [CB06a, CB06b] we showed that the detection of conflicts between concurrent instances of a composite collaboration can be automatized. This is not elaborated further here, and we just explain the main lines behind the proposed analysis approach. It is based on the fact that a component participates in a composite collaboration instance by playing a certain sequence of sub-roles (i.e. one sub-role for each sub-collaboration in the collaboration choreography the component participates in). When the same component participates in several collaboration instances, it plays several sequences of sub-roles. Undesired interactions may then arise between sub-roles belonging to different sequences (i.e. played in different collaboration instances) due to shared resources. In the proposed analysis approach, collaboration pre-conditions are used to specify the status and availability of the resources needed to execute a collaboration. Post-conditions describe the status and availability of resources after the collaboration execution. The analysis of interactions between the sequences of sub-roles played by a component is performed by constructing all possible interleavings of such sequences. If an interleaving contains two consecutive sub-roles such that the post-condition of the first one contradicts/falsifies the pre-condition of the second one, an interaction is reported.

5 Algorithms

In the following we present algorithms for the detection of race conditions and choice propagation problems. The input for these algorithms is a slightly modified version of the choreography graph. In a choreography an activity may describe two or more alternative behaviors, each one of them given by a different poset. Depending on the behavior that is executed, a different output pin may be used to pass on the focus of control to another activity (see left side of Fig. 12). Activities with these characteristics are replaced with a choice node and a set of new activities. Each new activity will be associated to one of the posets of the original activity. The interconnection of these new activities with the other activities in the graph is made according to the original interconnections (see right side of Fig. 12).

We also assume that the choreography graph does not contain interrupting and invocation compositions.

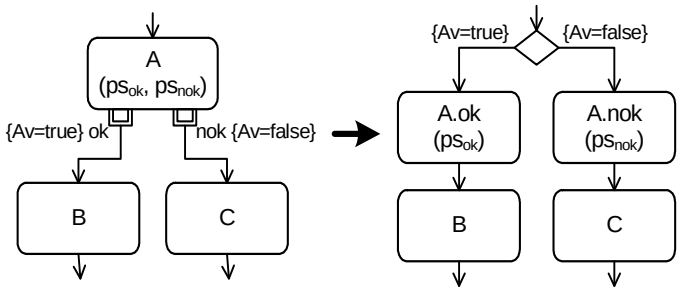

Figure 12: Replacement of collaborations with behavior described by several posets

5.1 Detection of Race Conditions

For the detection of races it is useful to know when a role p *gets synchronized* at a sending event s, so that no races may happen at p between a visual predecessor of s and any of its visual successors. Formally, given a visual order $<$ and its associated causal order $\prec$, we say that a role p *gets synchronized* at a sending event $s \in S_p$ if for any $e_1 \in \{e \in E_p : e <_p s\}$ and any $e_2 \in \{e \in E_p : s <_p e\}$, it holds that $e_1 \prec e_2$.

In an MSC, a race condition exists when two events are related by the MSC's visual order but not by its causal order. Building the visual and causal orders requires a transitive closure operation that has quadratic complexity on the number of events. Unfortunately, the detection of races in HMSCs is not so simple. Since an HMSC describes a collection of possible scenarios, it may happen that a race exists in one execution of the HMSC but it is compensated in another execution. That is, two events may be said to happen in one order in one of the HMSCs executions, and in the opposite order in another execution. As a result, the events are not in race. For the general class of HMSCs, the race detection problem is undecidable [MP00, Mus00].

A choreography of collaborations describes exact behavior, rather than existential behavior. Therefore, a race condition will exist in a choreography if it exists in any of its executions. The straightforward approach for the detection of races in a choreography requires to construct the visual and causal orders for all possible execution sequences, which might be computationally costly. In the following we present a set of algorithms for the detection of race conditions that do not require building the visual and causal orders for all possible execution sequences. We assume that all elementary collaborations are send-causal, that is, that their behaviors are described by sequence diagrams satisfying the send-causality property.

In the main algorithm (see Algorithm 1), we first look for races between events of the same elementary collaboration. For each elementary collaboration c, we construct the causal order of its events ($\prec^c$). We then check, for each role p, if there exist any pair of receiving events that are ordered by the total order associated to p (i.e. $r_1 <_p r_2$) but they are unordered according to the causal order of the collaboration (i.e. $r_1 \not\prec^c r_2$). If that is the case, a race exist between r_1 and r_2 (lines 1-4 of Algorithm 1). In case the sequence diagram of the elementary collaboration

contains loops, we consider just one iteration of the loop. If an event inside a loop is in race, we proceed according to the *GetRaceType* procedure (see page 41). Note that we only check for races between receiving events. This is because we assume that all elementary collaborations are send-causal, and according to Proposition A.3, a receiving event cannot be in race with a sending event when the send-causality property is satisfied. We may have actually used the results of Propositions A.5 and A.6 to detect race conditions without constructing the causal order of each elementary collaboration. We nevertheless built the causal orders since they will be needed in other parts of the race detection process.

Once all elementary collaborations have been analyzed, we search for races in the choreography graph, between events that belong to different collaborations. We propose two techniques to detect races in a choreography graph. One is used to detect races when the weak sequencing of collaborations in the choreography graph is send-causal, while the other is used when the weak sequencing is weakly-causal. We discuss these two techniques in the following.

Algorithm 1: DetectRaces

Data: A composite collaboration C; A choreography graph (V,E)
Result: A table *EventsInRace* containing in position (e_1,e_2) a set of collaboration sequences that lead to a race between events e_1 and e_2

```
1 foreach elementary sub-collaboration c of C do
2     Construct causal order ≺^c for c
3     foreach role p of c and each pair of receiving events r1, r2 ∈ E^c_p do
4         if r1 <_p r2 ∧ r1 ⊀^c r2 then EventsInRace[r1][r2] ← {c}
5 DetectRacesWithSendCausality()
6 DetectRacesWithWeakCausality()
7 DetectRacesInChainedActSeqs()
```

5.1.1 Races with Send-causal Weak Sequencing

When the weak sequencing of collaborations is send-causal, no sending event may be involved in a race condition (see Propositions A.1 and A.3). Only receiving events may be in race with other receiving events. Moreover, based on Propositions A.5 and A.6, only the visual order of events specified for a role p (i.e. $<_p$) needs to be considered in order to detect races between the receiving events performed by that role. That is, with send-causality the detection of races can be performed on a per-role basis, without taking into account the global causal order.

Propositions A.5 and A.6 show that two receiving events from the same role p, r_1 and r_2, may be in race only if, according to p's visual order, p does not perform any sending event between the two receiving events (i.e. if $r_1 <_p r_2$, and there is not a sending event s such that $r_1 <_p s <_p r_2$). Therefore, if a choreography describes a sequence of activities[9] $v_1 \cdot v_2 \cdot \ldots \cdot v_n$ $(1 < n)$, a race between $r_1 \in R_p^{v_1}$ (i.e. a receiving

[9]For the sake of simplicity we will use the terms activity and (sub-)collaboration interchangeably, since activities in the choreography refer to occurrences of sub-collaborations

event performed by role p in v_1) and $r_n \in R_p^{v_n}$ may only be possible if the two following conditions are satisfied:

- p plays a terminating sub-role in v_1 (i.e. p finishes its participation in v_1 with a receiving event) and a non-initiating sub-role in v_n (i.e. p starts its participation in v_n with a receiving event); and
- for each activity v_i, with $1 < i < n$, either p does not participate in v_i (i.e. $R_p^{v_i} = \emptyset$) or p only executes receiving events in v_i (i.e. $S_p^{v_i} = \emptyset$).

We propose to detect races in two steps. First, algorithm *DetectRacesWithSend-Causality* (on page 43) traverses the choreography graph and finds activity sequences of the form $v_1 \cdot v_2 \cdot \ldots \cdot v_n$ $(n > 1)$, where $v_1 \circ_w v_2$ is send-causal[10] and p only participates in v_1, playing a terminating sub-role, and in v_n, playing a non-initiating sub-role in v_n. For each of these sequences, a check is performed to detect potential races between v_1 and v_n at role p[11]. Thereafter, algorithm *DetectRacesInChainedActSeqs* (on page 57) tries to "chain" the activity sequences obtained by *DetectRacesWith-SendCausality*. That is, given a sequence of activities $v_1 \cdot v_2 \cdot \ldots \cdot v_n$ $(n > 1)$, where p has no sending event in v_n[12], the algorithm looks for any other sequences that starts with v_n. If a sequence $v_n \cdot v_{n+1} \cdot \ldots \cdot v_m$ $(m > n)$ is found, a check is performed to find races between the events of v_1 and v_m. After that, the process starts again, taking now the concatenated sequence, that is, $v_1 \cdot v_2 \cdot \ldots \cdot v_n \cdot v_{n+1} \cdot \ldots \cdot v_m$, as the initial sequence. To better understand the detection process, consider the sequence diagram in Fig. 13, which illustrates a sequence of collaborations[13]. For role $R2$, the *DetectRacesWithSendCausality* algorithm would find two sequences of collaborations with the aforementioned characteristics, namely $v_1 \cdot v_2 \cdot v_3$ and $v_3 \cdot v_4$ (note that $R2$ plays a terminating sub-role in v_4, but $v_4 \circ_w v_5$ is not send-causal). It would then check for races between the events of v_1 and v_3, and between the events of v_3 and v_4. Assuming communication channels with in-order delivery, a race would be detected between $?m_4$ (in v_3) and $?m_6$ (in v_4). The *DetectRacesInChainedActSeqs* algorithm would determine that sequences $v_1 \cdot v_2 \cdot v_3$ and $v_3 \cdot v_4$ can be chained (note that $R2$ has no sending event in v_3). It would then check for races between the events of v_1 and v_4. As a result, a race between $?m_1$ (in v_1) and $?m_6$ (in v_4) would be detected.

In the following we explain in more detail each of the procedures that are used as part of the *DetectRacesWithSendCausality* algorithm.

Algorithm *DetectRacesWithSendCausality* (on page 43). This algorithm performs a separate analysis of the choreography graph for each role p that may be

[10] The sequencing of the other activities might be either send-causal or weakly-causal.

[11] For the sake of brevity, we will talk about races between two activities v and w at a role p. This should be understood as races between an event $e_v \in E_p^v$ and an event $e_w \in E_p^w$.

[12] This is a requisite to chain sequences obtained by the *DetectRacesWithSendCausality* algorithm. However, as we will see, sequences where p performs a sending event in v_n may be chained with sequences obtained by the *DetectRacesWithWeakCausality* algorithm.

[13] The dashed rounded-rectangles are included just for illustration purposes, but are not standard UML

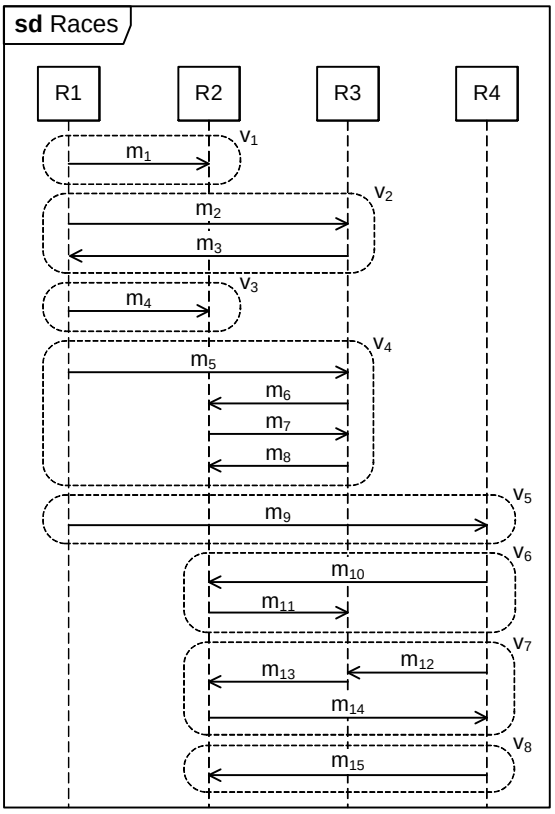

Figure 13: Races with send-causal and weakly-causal compositions

subject to potential races, that is, any role that plays a terminating sub-role in at least one sub-collaboration, and a non-initiating sub-role in at least other (possibly the same) sub-collaboration. Roles that play terminating sub-roles, but do not play non-initiating sub-roles, or roles that do not play any terminating sub-role are not subject to races[14], so they are not considered during the analysis of the choreography. This information can be obtained from the collaboration diagram.

For each role p that is subject to potential races, and from each activity v_1 where p plays a terminating sub-role, the algorithm invokes the *VisitSuccessorSC* procedure (on page 44) to perform a depth-first search (DFS) [AHU74] on the choreography graph. The idea is to find sequences of activities $v_1 \cdot v_2 \cdot \ldots \cdot v_n$ $(n > 1)$ where p only participates in v_1 and v_n and, thereby, check if there is any race between v_1 and v_n at role p.

Procedure *VisitSuccessorSC* (on page 44). This is a recursive procedure that performs a depth-first search [AHU74] on the choreography graph to find sequences of activities $v_1 \cdot v_2 \cdot \ldots \cdot v_n$ $(n > 1)$ where p only participates in v_1 and v_n. Once such a sequence is found, that is, once an activity v_n where p participates is found, procedure *CheckRacesSC* is invoked (line 5). This procedure checks whether

[14]They may actually have races inside a given collaboration, but not between two collaborations.

there is any race between v_1 and v_n at role p (see details on page 41). When *CheckRacesSC* returns, the backtracking process is initiated until a node with unvisited successors is found.

We note that, for a given activity v_1 and a role p, a possibly infinite number of activity sequences $v_1 \cdot v_2 \cdot \ldots \cdot v_n$ ($n > 1$), where p only participates in v_1 and v_n, may be found in the choreography graph. Fortunately, only a subset of these sequences is of interest for our purposes. We note the following:

1. In order to detect a race between two activities v_1 and v_n at role p, it is sufficient to find one sequence of the form $v_1 \cdot v_2 \cdot \ldots \cdot v_n$ ($n > 1$), where p only participates in v_1 and v_n. This has a positive implication on loops: we just need to traverse the body of a loop once, thus avoiding infinite sequences. Therefore, during each traversal of the graph, each activity is visited as most once. For example, in the case of the choreography in Fig. 14(a) the algorithm would only return the sequence $v_1 \cdot v_2 \cdot v_3$, which is sufficient to detect races between v_1 and v_3.
2. Given a sequence of activities $v_1 \cdot v_2 \cdot \ldots \cdot v_n$, a race between v_1 and v_n is only possible if v_1 and v_2 are composed in weak sequence. Making strong the sequencing between v_1 and v_2 would eliminate any potential race between v_1 and v_n, but only when v_n is reached via v_2. The race may still be possible if v_n is reached through another path. This means that, given two activities v_1 and v_n, we are interested in the set of all sequences that start at v_1, end at v_n, and have a second activity that is different from the second activity of any other sequence in the set. For example, in the choreography of Fig. 14(c) we are interested in two sequences, namely $v_1 \cdot v_2 \cdot v_3$ and $v_1 \cdot v_2' \cdot v_3$, while in the choreography of Fig. 14(b) we are only interested in one sequence, either $v_1 \cdot v_2 \cdot v_3 \cdot v_5$ or $v_1 \cdot v_2 \cdot v_4 \cdot v_5$.
   To obtain the set of desired sequences, procedure *VisitSuccessorSC* allows decision nodes to be revisited multiple times, while merge nodes may be revisited only in certain cases. Whenever a merge node is visited, its *visited* flag is set to *true*. In a traditional DFS algorithm, such flag would be reset during backtracking. This does not happen in the proposed algorithm (line 45). Instead, when a decision node is visited, and before visiting any of its successor nodes, the algorithm checks whether *AuxSeq* (i.e. the ordered set of visited activities in the current path) contains only one element (i.e. v_1). If that is the case, a different "second activity" will be visited. It is then that the *visited* flag of merge nodes is set to *false* (lines 13-15), so that v_n can be visited again (if that is possible through the new path). For example, in the choreographies of Figs. 14(a) and 14(b), the *visited* flag of m_1 would not be reset when visiting d_1, since $AuxSeq = \{v_1, v_2\}$ at that point. In the case of the choreographies in Figs. 14(c) and 14(d), $AuxSeq = \{v_1\}$ when visiting d_1, so the *visited* flag for m_1 would be reset.

The treatment of fork and join nodes by the *VisistSuccessorSC* procedure deserves some explanation. In the following we assume that fork and join nodes are properly nested. That is, all outgoing edges of a fork node lead to the same join node (in the following called the *companion join*), and all incoming edges of a join

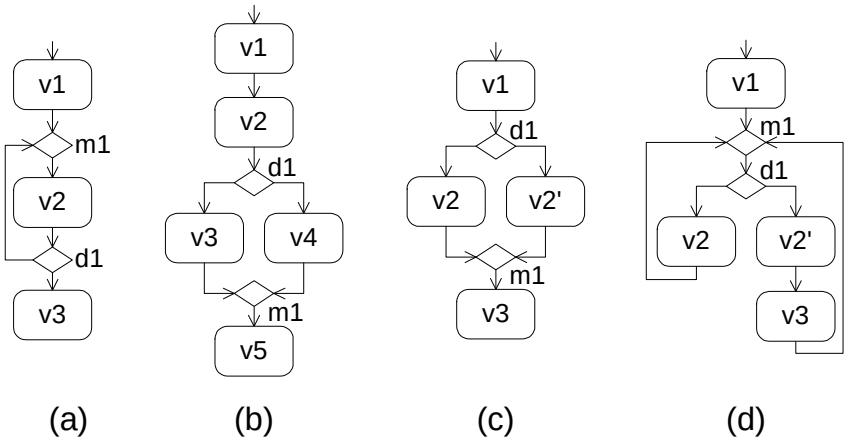

Figure 14: Examples of combinations of merge and decision nodes in a choreography graph

node come from the same fork node[15]. We note that each branch of a fork (corresponding to each of the fork's outgoing edges) may define several execution paths. For example, in Fig. 15 the fork's right branch defines two execution paths, namely $v_4 \cdot v_5$ and $v_4 \cdot v_6$.

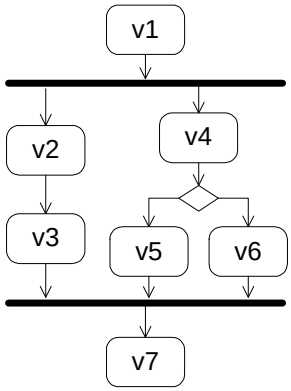

Figure 15: Properly nested fork and join nodes in a choreography graph

When a fork node is visited, a search is performed in each of the fork's branches for activities where p participates. This is done by invoking the *TraverseForkBranch* procedure for each of the fork's successor nodes (line 25). This procedure returns three values:

- *continue*: it is a boolean predicate that is *true* when p does not participate in one or more (possibly all) of the execution paths of the traversed branch,

[15] An exception is the following. If the join node is connected to a final node, the former could be removed, and let all outgoing edges of the fork lead to final nodes.

and gets synchronized[16] in the remaining paths. For example, consider that p does not participate in any of the activities inside the fork-join pair in Fig. 15. Then *continue* would be *true* for both branches of the fork. If p only participates in v_6 and gets synchronized, *continue* would still be *true* for the right branch, but if it does not get synchronized, then *continue* would be *false* for the right branch.

- *auxSynch*: it is a boolean predicate that is *true* when p gets synchronized in all execution paths of the traversed branch.

- *lastPMRs*: it is a set containing, for each of the execution paths of the traversed branch, the index of the last entry created in $PMRSeqs_{\text{sc}}$. Note that *TraverseForkBranch* invokes *VisitSuccessorSC*, which may in turn invoke *CheckRacesSC*, where new entries can be added to the $PMRSeqs_{\text{sc}}$ table.

Once all the branches of the fork has been traversed, and if *contAfterFork* is *true* (this only happens if *continue* was *true* for all the fork's branches – see line 28), *VisitSuccessorSC* continues traversing the choreography graph from the fork's companion join node (line 41). Otherwise, if *contAfterFork* is *false*, *VisitSuccessorSC* starts the backtracking process. This may happen in two cases:

a) p gets synchronized in one or more of the fork's branches (i.e. *auxSynch* is *true* for each of those branches and *synch* is *true* – see line 27). No races are then possible between activities that precede the fork and activities that succeed the fork's companion join. In this case the *DetectRacesInChainedActSeqs* algorithm should not try to chain activity sequences whose first activity precedes the fork with activity sequences whose last activity succeeds the fork's companion join. For example, consider that, after analysing the graph in Fig. 15, *VisitSuccessorSC* stores three activity sequences in $PMRSeqs_{\text{sc}}$, namely $v_1 \cdot v_2$, $v_2 \cdot v_3$ and $v_3 \cdot v_7$. Assume also that p gets synchronized in the fork's right branch. Now, the *DetectRacesInChainedActSeqs* algorithm would chain the $v_1 \cdot v_2$ and $v_2 \cdot v_3$ sequences, but should not try to chain the resulting sequence (i.e. $v_1 \cdot v_2 \cdot v_3$) with $v_3 \cdot v_7$. This is because p gets synchronized in one of the fork's branches, and no races can thus happen between events of $v_1$and events of v_7. To ensure that *DetectRacesInChainedActSeqs* behaves as expected, the last entries created in $PMRSeqs_{\text{sc}}$ for each of the fork's branches, whose indexes are stored in *lastPMRs*, are "marked". In the previous example the entry containing $v_2 \cdot v_3$ in would be marked. This is done with help of the table *synchPMR*, which associate each element in *lastPMRs* with the first activity in *AuxSeq*, which precedes the fork (lines 32-34).

b) p does not get synchronized in any of the fork's branches, and p participates in at least one of the activities of one of the fork's branches. In this case races

[16] A role gets synchronized at a sending event if no races may happen between events preceding and succeeding that sending event. See the explanation of procedure *GetsSynchronized* on page 39 for more details.

between activities preceding the fork and activities succeeding the fork's companion join are possible, but will be detected by the *DetectRacesInChainedActSeqs* algorithm. Consider the example in Fig. 15 and imagine that p plays initiating sub-roles in v_1, v_2 and v_7 (p does not participate in the other activities). Starting from v_1, *VisitSuccessorSC* would traverse the fork's left branch, find the sequence $v_1 \cdot v_2$, and check for races between $v_1$and v_2. Then it would traverse the fork's right branch. After that, since *continue* would be *false* for the left branch, it would start the backtracking process. Thereafter, starting from v_2, *VisitSuccessorSC* would find the sequence $v_2 \cdot v_3 \cdot v_7$, and check for races between $v_2$and v_7. Potential races between v_1 and v_7 would be found by the *DetectRacesInChainedActSeqs* algorithm when chaining the two sequences $v_1 \cdot v_2$ and $v_2 \cdot v_3 \cdot v_7$.

Procedure *TraverseForkBranch* (on page 45). This procedure first invokes the *VisitSuccessorSC* procedure in order to find an activity in a fork's branch where a role p participates. There situations can then be differentiated:

- p does not participate in any of the execution paths of the fork's branch. Then *joinFound* will be *true* and $k = 0$. *TraverseForkBranch* will thus return with $(true, false, \emptyset)$.
- p does not participate in some of the execution paths of the fork's branch, and gets synchronized in the other paths. Then *joinFound* will be *true* and $k > 0$. *TraverseForkBranch* will again return with $(true, false, \emptyset)$, since F will be an empty set.
- p participates in some of the execution paths. For each of those paths an entry in the $PMRSeqs_{\text{sc}}$ table should have been created by the *CheckRacesSC* procedure, and the index of such entry should have been stored in the *forkPMR* array. Then, for each entry in $PMRSeqs_{\text{sc}}$, *TraverseForkBranch* is recursively invoked, but this time starting from the successor node of the last activity of the activity sequence stored in $PMRSeqs_{\text{sc}}$. As a result of the recursion, all the execution paths of the branch will be traversed until either the fork's companion join node or an activity where p gets synchronized is found. At the end of this process, *lastPMRs* will contain the index of the last entry created in $PMRSeqs_{\text{sc}}$ in each of the execution paths. *synch* will be *true* if p got synchronized in all execution paths, and *contAfterFork* will be *true* if p did not participate in one or more (possibly all) of the execution paths, and got synchronized in the remaining paths.

Procedures *GetsSynchronized* (on page 47) and *SynchronizedRoles* (on page 47). We say that a role p *gets synchronized* at a sending event s, if no races may happen at p between a visual predecessor of s and any of its visual successors. More formally, given a visual order $<$ and its associated causal order $\prec$, we say that a role p *gets synchronized* at a sending event $s \in S_p$ if for any $e_1 \in \{e \in E_p : e < s\}$ and any $e_2 \in \{e \in E_p : s < e\}$, it holds that $e_1 \prec e_2$. Intuitively,

procedure *GetsSynchronized* determines whether a role p gets synchronized, at a specific point, in some of the execution paths described by the choreography graph.

Consider that the choreography describes a sequence of activities $v_1 \cdot \ldots \cdot v_i \cdot v_{i+1} \cdot \ldots$ $(i > 0)$. In general, this sequence of activities defines several alternative execution paths, since each activity may describe several alternative behaviors. Each of those execution paths is represented by a visual order $(E, <)$ and a causal order $(E, \prec) = ps_1 \circ_w \ldots \circ_w ps_i \circ_w ps_{i+1} \circ_w \ldots$ (where each ps_i is one of the causal partial orders associated with activity v_i). A given activity may appear in several sequences of activities, and each of these sequences may define several execution paths. Given an activity v_i, a causal order $ps_i = (E_i, \prec_i)$, and a role p, procedure *GetsSynchronized* returns *true* if p gets synchronized at a sending event $s \in E_i$ in all the choreography's execution paths that contain the behavior described by ps_i. Otherwise, it returns *false*.

We explain in the following the process of checking whether a role gets synchronized in a given execution path whose visual order is $(E, <)$ and its causal order is $(E, \prec) = ps_1 \circ_w \ldots \circ_w ps_i \circ_w ps_{i+1} \circ_w \ldots \circ_w ps_n$. Recall that we assume that the sequential composition of activities in the choreography graph is weakly-causal, and that each individual activity is send-causal. This means that any ps_i in $(E, \prec)$ is send-causal, and any $ps_i \circ_w ps_{i+1}$ is weakly-causal. We consider initially the case where activities describe sequential behaviors. Later, we generalize the result to consider the possibility of concurrent behaviors.

Assume that role p executes at least one sending event in the behavior described by ps_i (otherwise p does not get synchronized). Let $s_p \in E_i$ be the minimum sending event of role p in ps_i (i.e. there is no other sending event $s' \in E_i$ such that $s' \prec_i s_p$). Consider now two events located at p, e_1 and e_2, such that $e_1 < s_p$ (i.e. e_1 is a visual predecessor of s_p) and $s_p < e_2$ (i.e. e_2 is a visual successor of s_p). We know that $e_1 \prec s_p$, since a role will not execute a sending until all events that are specified to happen before that sending have been processed. Therefore, to determine whether $e_1 \prec e_2$ (and thereby determining whether p gets synchronized at s_p), we just need to check whether $s_p \prec e_2$. Two cases can be differentiated. If $e_2 \in E_u$, given that ps_i is send-causal, and according to Proposition A.3, we have that $s_p \prec e_2$. If otherwise $e_2 \notin E_i$, then it should be the case that $e_2 \in E_{i+1} \cup \ldots \cup E_n$. Let us take a closer look at this case. Let s_j^{min} be the minimum sending event in ps_j $(1 \leq j \leq n)$. Let now q be the initiating role of ps_{j+1}, that is, the role executing the minimum sending event in ps_{j+1} (i.e. $loc(s_{j+1}^{min}) = q$), and let e_j^{max-q} be the maximum event in ps_j of q[17]. Since ps_j is send-causal we have that $s_j^{min} \prec e$, for any $e \in E_j$, and therefore $s_j^{min} \prec e_j^{max-q}$ or $s_j^{min} = e_j^{max-q}$. Now, since $e_j^{max-q} \prec s_{j+1}^{min}$ (due to the definition of weak sequencing), we have that $s_j^{min} \prec s_{j+1}^{min} \prec e'$, for any $e' \in E_{j+1}$. It is easy then to see that $s_j^{min} \prec e_x$, for any $e_x \in E_j \cup \ldots \cup E_n$ and any $j \in \{1 \ldots n\}$. In particular, it must be the case that $e_i^{max-q} \prec s_{i+1}^{min} \prec e_2$, where $loc(e_i^{max-q}) = loc(s_{i+1}^{min}) = q$. Therefore, to determine whether $s_p \prec e_2$, we just need to check whether $s_p \prec e_i^{max-q}$ or $s_p = e_i^{max-q}$. This is what procedure *SynchronizedRoles* does.

[17] We note that q will always participate in ps_j, since $ps_j \circ_w ps_{j+1}$ is weakly-causal

Procedure *SynchronizedRoles* takes also into account that the causal orders may describe concurrent behaviors. In general, p may have a set Min of minimum sending events in ps_i. There might also be a set I of initiating roles in ps_{i+1}, and each of them may have a set Max_q of maximum events in ps_i. Then, in order for role p to get synchronized, it is necessary that for each role $q \in I$, there is at least one minimum sending $s_p \in Min$ and one maximum event $e_i^{max-q} \in Max_q$ such that $s_p \prec e_i^{max-q}$ or $s_p = e_i^{max-q}$. This ensures that $s_p \prec s_{i+1}^{min}$ for each role q.

Procedure *GetRaceType* (on page 47). In case the behavior of an activity contains loops, procedure *CheckRacesSC* considers only one iteration for each of the loops. When a race is detected between two events and one of the events, or both, are inside a loop, a question arises whether all potential instances of those events (when several loop iterations are considered) will be in race. Procedure *GetRaceType* answers that question by classifying a race as *type1* and/or *type2*. This procedure assumes send-causality (i.e. for any loop, the sequential composition of its body with itself, and with the preceding and succeeding behaviors, is send-causal) and, given a receiving event r_1 that is in race with another receiving event r_2 (i.e. $r_1 <_p r_2$ but $r_1 \not\prec r_2$), differentiates two main cases:

a) r_1 is a loop event. If there is a sending event inside all loops that contain r_1, then only the instance of r_1 corresponding to the last iteration of the loops will be in race with r_2. Otherwise, each instance of r_1 will be in race with r_2, and the race between r_1 and r_2 is said to be a *type1* loop-race (i.e. $type1 = true$).

b) r_2 is a loop event. If there is a sending event inside all loops that contain r_2, then r_1 is in race only with the instance of r_2 corresponding to the first iteration of the loops. Otherwise, r_1 will be in race with all instances of r_2, , and the race between r_1 and r_2 is said to be a *type2* loop-race (i.e. $type2 = true$).

Procedure *CheckRacesSC* (on page 46). This procedure checks whether there is any race at role p between events in v_1 (i.e. the first activity in the current sequence of activities) and events in u (i.e. the first activity after v_1 where role p participates). The procedure takes into account that activities may describe alternative behaviors, so each activity may have several associated posets, as well as concurrent behaviors, so p may have several minimum and maximum sendings in each poset. It also takes into account that activities may describe loops. Only one iteration of each loop is considered to build the visual orders used to detect races. The effect of multiple loop iterations is considered by invoking procedure *GetRaceType*.

For race detection the procedure obtains, for each poset of v_1 and each poset of u, the sets $R_{\text{race}}^{v_1}$ and R_{race}^{u} of receiving events that might be in race. $R_{\text{race}}^{v_1}$ (line 4) contains receiving events from v_1 that do not have any sending event as a successor, while R_{race}^{u} (line 13) contains receiving events from u that do not have any sending event as a predecessor. If non-FIFO channels are used for communication, all these events will be in race. Otherwise, if FIFO channels are used, a receiving event $r_1 \in R_{\text{race}}^{v_1}$ will be in race with a receiving event $r_2 \in R_{\text{race}}^{u}$ if their associated sending events are not located at the same role. Whenever a race is found between two

events r_1 and r_2, the value of *AuxSeq* (i.e. the current sequence of visited activities) is stored in table *EventsInRace* (line 18). In addition, procedure *GetRaceType* is invoked to determine whether the events that are in race are inside a loop and, if so, check whether all potential instances of those events (when several loop iterations are executed) are or not in race. The result of this procedure is stored in the *RaceType* table (line 19).

In addition to checking the existence of races between v_1 and u, procedure *CheckRacesSC* determines whether races may exist at role p between v_1 and other activities that may be executed after u. This information will then be used by procedure *TraverseForkBranch* and algorithm *DetectRacesInChainedActSeqs*.

Races between v_1 and any successor activity of u might be possible in the following three cases:

a) If u describes several alternative behaviors, and p does not participate in some of them (line 6).

b) If p only executes receiving events in one of the possible behaviors described by u (line 20).

c) If p executes a sending event in one of the possible behaviors described by u, but it does not get synchronized at that sending event[18] (line 24).

In the three cases above, procedure *CheckRacesSC* stores in the $\boldsymbol{PMRSeqs}_{\mathrm{sc}}$ table data that will be used by *DetectRacesInChainedActSeqs* for the actual detection of races (namely $\boldsymbol{AuxSeq}, \boldsymbol{R}^{v_1}_{\mathrm{race}}, \boldsymbol{R}^{u}_{\mathrm{race}}$ and a boolean value specifying whether there was any race between events of v_1 and u). In addition, the index of the entry created in $\boldsymbol{PMRSeqs}_{\mathrm{sc}}$ is stored in the *forkPMR* set and in the $\boldsymbol{PMR}_{\mathrm{rcv}}$ set (in cases a and b) or in the $\boldsymbol{PMR}_{\mathrm{snd}}$ set (in case c). Also in the three cases above, a boolean entry in the *synchSB* array is created and set to *false*, meaning that role p does not get synchronized in the current execution path. Otherwise, if no one of the three above cases applies, an entry in the *synchSB* array is created and set to *true*, meaning that p gets synchronized in the current execution path.

[18] Note that this implies that the sequential composition of u with one of its succeeding activities in the choreography is weakly-causal.

Algorithm 2: DetectRacesWithSendCausality

Data: A choreography graph (V,E)

Result: A table *EventsInRace* containing in position (e_1,e_2) a set of collaboration sequences that lead to a race between events e_1 and e_2; Sets *PMR* and *PMRSeqs* that are useful for the detection of races involving several collaboration sequences

```
1 foreach role p playing both terminating and non-initiating sub-roles do
2     foreach v1 ∈ V where p plays a terminating sub-role do
3         forall v ∈ V do visited[v] ← false
4         i ← 0; VisitedMerge ← ∅
5         AuxSeq ← {v1} // Ordered sequence of activities
6         v ← successor of v1
7         VisitSuccessorSC(v, false, false, ∅, 0)
```

Procedure `VisitSuccessorSC`(v, forkFound, joinFound, *synchSB*, k)

```
1  visited[v] ← true
2  if v is an activity node then
3      AuxSeq ← AuxSeq ∪ {v}
4      if p participates in v then
5          (synchSB, k) ← CheckRacesSC(v, synchSB, k)
6          AuxSeq ← AuxSeq − {v}; visited[v] ← false // Backtrack
7          return (joinFound, synchSB, k)
8  else if v is a merge node then
9      VisitedMerge ← VisitedMerge ∪ {v}
10 else if v is a decision node then
11     visited[v] ← false // Decision nodes can always be revisited
12     foreach u successor of v do
13         if |AuxSeq| = 1 then
14             forall w ∈ VisitedMerge do visited[w] ← false
15             VisitedMerge ← ∅
16         if !visited[u] then
           (joinFound, synchSB, k) ← VisitSuccessorSC(u, forkFound, joinFound, synchSB, k)
17     return (joinFound, synchSB, k)
18 else if v is a join node ∧ forkFound then
19     v_join ← v; visited[v] ← false
20     return (true, synchSB, k) // joinFound is set to true
21 else if v is a fork node then
22     forkFound ← true; v_join ← null
23     synch ← false; contAfterFork ← true; allLastPMRs ← ∅; AuxSeq_old ← AuxSeq
24     foreach successor u of v do
25         (continue, auxSynch, lastPMRs) ← TraverseForkBranch(u, ∅, ∅)
26         allLastPMRs ← allLastPMRs ∪ lastPMRs
27         synch ← synch ∨ auxSynch
28         contAfterFork ← contAfterFork ∧ continue
29         AuxSeq ← AuxSeq_old
30     forkFound ← false; joinFound ← false
31     if !contAfterFork then
32         if synch = true then
33             v_1 ← first element of AuxSeq
34             forall (u, p, j) ∈ allLastPMRs do synchPMR[u][p][j] ← v_1
35             synchSB[k] ← true; k++
36         else
               // In case the just visited fork is inside another fork-join pair
37             foreach (u, p, j) ∈ allLastPMRs do
38                 forkPMR[k] ← (u, p, j); synchSB[k] ← false; k++
39         visited[v] ← false // Backtrack
40         return (false, synchSB, k)
41     AuxSeq ← AuxSeq ∪ {v}
42     v ← v_join // Continue traversing graph from companion join
43 u ← successor of v
44 if !visited[u] then
   (joinFound, synchSB, k) ← VisitSuccessorSC(u, forkFound, joinFound, synchSB, k)
   /* Backtrack                                                              */
45 if v is NOT a merge node then visited[v] ← false
46 return (joinFound, synchSB, k)
```

Procedure `TraverseForkBranch`$(v, prevPMR, lastPMRs)$

```
   /* forkPMR and PMRSeqs_sc are global variables whose data is set in the
   CheckRacesSC procedure                                                    */
 1 synch ← true; contAfterFork ← false
 2 (joinFound, synchSB, k) ← VisitSuccessorSC(v, true, false, ∅, 0)
 3 if joinFound then synch ← false
 4 if joinFound ∧ prevPMR = ∅ then contAfterFork ← true
 5 if !joinFound ∨ (joinFound ∧ k > 0) then
 6     lastPMRs ← lastPMRs − prevPMR
 7     F ← {forkPMR[j] : j ∈ 0...k−1 ∧ synchSB[j] = false}
 8     foreach (w, q, j) ∈ F do
 9         lastPMRs ← lastPMRs ∪ {(w, p, j)}
10         u ←last activity of PMRSeqs_sc[w][q][j].AuxSeq
11         AuxSeq ← {u}; i_old ← i; i ← 0
12         x ←successor of u
13         (continue, auxSynch, lastPMRs) ← TraverseForkBranch(x, {(w, q, j)}, lastPMRs)
14         i ← i_old
15         synch ← synch ∧ auxSynch
16         if !auxSynch then contAfterFork ← false
17 return (contAfterFork, synch, lastPMRs)
```

Procedure `CheckRacesSC`$(u, synchSB, k)$

Data: Activity u with which v_1 (first act. in *AuxSeq*) could be in race

Result: *EventsInRace* and *RaceType* are updated if a race is found; Entries in $PMR_{\text{rcv/snd}}$ and $PMRSeqs_{\text{sc}}$ are created if v_1 might be in race with an activity following u (to be used by the *DetectChainRaces* algorithm); Entries in $forkPMR$ and *synch* are created for use in the *TraverseForkBranch* procedure

1 $v_1 \leftarrow$ first element of *AuxSeq*
2 **foreach** *visual order* $(E_p^{v_1}, <_p^{v_1})$ *of* v_1 **do**
3 $S_{\max}^{v_1} \leftarrow \{s \in S_p^{v_1} : \nexists s' \in S_p^{v_1}, s <_p^{v_1} s'\}$ `// max sending events in` $ps_p^{v_1}$
`/* Obtain set` $R_{\text{race}}^{v_1}$ `of receiving events from` $ps_p^{v_1}$ `that could be in race (i.e. all receiving events, except those that according to` $<_p^{v_1}$ `precede any of the maximum sending events) */`
4 $R_{\text{race}}^{v_1} \leftarrow R_p^{v_1} - \{r \in R_p^{v_1} : \exists s_{\max} \in S_{\max}^{v_1}, r <_p^{v_1} s_{\max}\}$
5 **foreach** *visual order* $(E_p^u, <_p^u)$ *of* u **do**
6 **if** $E_p^u = \emptyset$ **then** `/*` p `does not participate in this alternative of` u `*/`
7 $PMRSeqs_{\text{sc}}[v_1][p][i] \leftarrow (AuxSeq, R_{\text{race}}^{v_1}, \emptyset, false)$; $PMR_{\text{rcv}} \leftarrow PMR_{\text{rcv}} \cup \{(v_1, p, i)\}$
8 $i{+}{+}$
9 $forkPMR[k] \leftarrow (v_1, p, i)$; $synchSB[k] \leftarrow false$; $k{+}{+}$
10 **else**
11 $race \leftarrow false$
12 $S_{\min}^u \leftarrow \{s \in S_p^u : \nexists s' \in S_p^u, s' <_p^u s\}$ `// min sending events in` ps_p^u
`/* Obtain set` R_{race}^u `of receiving events from` ps_p^u `that could be in race (i.e. all receiving events, except those that according to` $<_p^u$ `happen after any of the minimum sending events) */`
13 $R_{\text{race}}^u \leftarrow R_p^u - \{r \in R_p^u : \exists s_{\min} \in S_{\min}^u, s_{\min} <_p^u r\}$
`// Check races`
14 **foreach** $r_{v_1} \in R_{\text{race}}^{v_1}$ **do**
15 **foreach** $r_u \in R_{\text{race}}^u$ **do**
16 **if** *non-FIFO OR (FIFO AND* $loc(snd(e_{v_1})) \neq loc(snd(e_u))$*)* **then**
17 $race \leftarrow true$
18 $EventsInRace[r_{v_1}][r_u] \leftarrow EventsInRace[r_{v_1}][r_u] \cup \{AuxSeq\}$
`/* Check if the race involves events inside loops */`
19 $RaceType[r_{v_1}][r_u] \leftarrow GetRaceType(r_{v_1}, r_u, S_{\max}^{v_1}, S_{\min}^u)$
20 **if** $S_{\min}^u = \emptyset$ **then** `//` p `has only receiving events in` E_p^u
21 $PMRSeqs_{\text{sc}}[v_1][p][i] \leftarrow (AuxSeq, R_{\text{race}}^{v_1}, R_{\text{race}}^u, race)$
22 $PMR_{\text{rcv}} \leftarrow PMR_{\text{rcv}} \cup \{(v_1, p, i)\}$; $i{+}{+}$
23 $forkPMR[k] \leftarrow (v_1, p, i)$; $synchSB[k] \leftarrow false$; $k{+}{+}$
24 **else if** *NOT* $GetsSynchronized(p, u, ps^u)$ **then**
25 $PMRSeqs_{\text{sc}}[v_1][p][i] \leftarrow (AuxSeq, R_{\text{race}}^{v_1}, R_{\text{race}}^u, race)$
26 $PMR_{\text{snd}} \leftarrow PMR_{\text{snd}} \cup \{(v_1, p, i)\}$; $i{+}{+}$
27 $forkPMR[k] \leftarrow (v_1, p, i)$; $synchSB[k] \leftarrow false$; $k{+}{+}$
28 **else**
`/* Role` p `gets synchronized. No possibility of race at` p `between` v_1 `and any activity following` u`, so no` *PMR* `and` *PMRSeqs* `entries needed. */`
29 $synchSB[k] \leftarrow true$; $k{+}{+}$
30 **return** $(synchSB, k)$

Procedure `GetRaceType`$(r_{v_1}, r_u, S^{v_1}_{max}, S^{u}_{min})$

```
1  (type1, type2) ← (false, false) // No special race
2  if r_{v1} is inside one or more nested loops then
       /* There will be one instance of r_{v1} for each loop iteration */
3      if ∄ s_max ∈ S^{v1}_max such that s_max is contained by all nested loops that contain r_{v1} then
           /* All instances of r_{v1} are in race with r_u */
4          type1 ← true
5  if r_u is inside one or more nested loops then
       /* There will be one instance of r_u for each loop iteration */
6      if ∄ s_min ∈ S^{u}_min such that s_min is contained by all nested loops that contain r_u then
           /* r_{v1} is in race with all instances of r_u */
7          type2 ← true
8  return (type1, type2)
```

Procedure `GetsSynchronized`(p, u, ps_u)

```
   Result: true if p gets synchronized for all successor posets of ps_u. false otherwise
1  result ← true
2  foreach successor activity w of u do
3      foreach poset ps_w of w do
4          if SynchronizedRoles(ps_u, ps_w, {p}) = ∅ then
5              result ← false
6  return result
```

Procedure `SynchronizedRoles`$(ps_1, ps_2, \mathscr{R})$

```
   /* We assume ps_1 = (S_1 ∪ R_1, ≺_1) */
1  R_synch ← ∅
   /* Get subset R_snd of roles from R that have a sending event in ps_1 */
2  R_snd ← {loc(s) : loc(s) ∈ R ∧ s ∈ S_1}
3  if R_snd ≠ ∅ then
4      if ps_1 ∘_w ps_2 is weakly-causal then
5          I ← {loc(e) : e ∈ min(ps_2)} /* Initiating roles of ps_2 */
6          foreach q ∈ I do
7              Max[q] ← {e : e ∈ max(ps_1) ∧ loc(e) = q} /* Max events in ps_1 of q role */
8          foreach p ∈ R_snd do
9              Min ← {s : s ∈ min((S_1, ≺_1)) ∧ loc(s) = p} /* Min sendings in ps_1 of p */
10             if ∀q ∈ I, ∃m ∈ Max[q], ∃s ∈ Min such that s ≺_1 m ∨ s = m then
11                 R_sych ← R_synch ∪ {p}
12     else /* Send-causal sequencing */
13         R_synch ← R_snd
14
15 return R_synch
```

5.1.2 Races with Weakly-causal Weak Sequencing

When the weak sequencing of collaborations is weakly-causal, the global causal order of events has to be considered for race detection. Consider again the example in Fig. 13. The sequential composition $v_4 \circ_w v_5$ is weakly-causal. It is easy to see that events happening in v_4 after $!m_5$ (i.e. the events performed by $R2$ and $R3$) may be in race with other events in v_5, v_6, v_7 and v_8. Let us focus on role R_2, which performs three events in v_4 after $!m_1$, namely $?m_6, !m_7$ and $?m_8$. Just by looking at the local ordering of events in the lifeline of $R2$ we cannot determine whether, for example, $?m_6$ is in race with $?m_{10}, ?m_{13}$ or $?m_{15}$. To find this out we need to consider the causal order between the events of $R2$ and the events performed by the other roles. Fortunately, the total number of events that we need to consider in order to build such causal order can be limited. In this case, for example, we do not need to consider the events of v_8 (and of any other collaboration that may succeed v_8) in order to detect races at $R2$. This is because $R2$ gets synchronized at sending event $!m_{14}$ in v_7 (see explanation on page 39), so no races may happen at $R2$ between events preceding and succeeding $!m_{14}$ in $R2$'s lifeline.

In the following we explain in more detail the *DetectRacesWithWeakCausality* algorithm and each of the procedures that are used by this algorithm.

Algorithm *DetectRacesWithWeakCausality* (on page 51). For each causal poset $ps_1 = (E_1, \prec_1)$ of an activity v_1, and each causal poset ps_2 of an activity v_2, such that $ps_1 \circ_w ps_2$ is weakly-causal, this algorithm finds race conditions involving events of E_1. For that, it first obtains the set $\mathscr{R}$ of roles that are subject to potential races, that is, roles whose events may be in race with other events (lines 4-9). Intuitively, these are roles that may execute an event from E_1 when the behavior described by ps_2 has already been started. Given an initiating-role q of ps_2, and a maximum event m of q in ps_1 (if m is a receiving event, its associated sending event is considered instead), a role may execute an event $e \in E_1$ after q has initiated ps_2 if e will always be executed after m (i.e. $m \prec_1 e$), or if e and m may be executed in any order (i.e. $m \not\prec_1 e$ and $e \not\prec m$). We note, however, that the algorithm only considers roles that may execute a sending event form ps_1 when ps_2 has already been initiated (line 9). This is because races affecting a role that may execute receiving events from ps_1, but not sending events, when ps_2 has already been initiated, are already detected by the *DetectRacesWithSendCausality* algorithm.

Once the set of roles subject to potential races has been obtained, the algorithm invokes the *VisitSuccessorWC* procedure to perform a depth-first search on the choreography graph. This procedure returns an array *SEQS* (via a global variable) whose elements are tuples of the form $(seq, ps, \mathscr{R}')$, where $seq = v_1 \cdot v_2 \cdot \ldots \cdot v_n$ is a sequence of activities, $ps = ps_1 \circ_w ps_2 \circ_w \ldots \circ_w ps_n$ $(n \geq 2)$ is the causal poset corresponding to one of the execution paths associated to seq, and $\mathscr{R}' \subseteq \mathscr{R}$ is the set or roles that did not get synchronized in any of the posets in ps. Normally, $\mathscr{R}'$ will be the empty set, unless a final node or an infinite loop was found in the choreography (see the explanation of *VisitSuccessorWC*).

After *VisitSuccessorWC* returns, the *CheckRacesWC* procedure is invoked to detect races with help of the causal posets previously obtained.

Procedure *VisitSuccessorWC* (on page 52). This is a recursive procedure that performs a depth-first search [AHU74] in the choreography graph.

Control nodes, as well the posets of activity nodes, can be "visited" at most twice. This avoids infinite sequences in the presence of loops, while it ensures that all combinations of activities that are interesting for the detection of races are considered.

Each time an activity node is visited, each one of its posets is visited. The procedure then checks whether any role in $\mathscr{R}$ got synchronized in the previously visited poset (when that poset is weak sequenced with the current one) and, if so, it removes the roles that got synchronized from $\mathscr{R}$ (line 7). When all roles in $\mathscr{R}$ get eventually synchronized (and if the *done* flag is *false*), the current sequence of visited activities, and the causal poset corresponding to the sequence of visited posets are stored in the *SEQS* array (by invoking the *ProcessSeq* procedure in line 10). The *done* flag is done to avoid having several equal entries in the *SEQS* array, in case $\mathscr{R}$ becomes empty for several posets of the same activity. If not all roles got synchronized, the current activity and poset are added to the sequences of visited activities and posets, and the traversal of the graph continues by recursively invoking *VisistSuccessorWC* (line 13). The backtracking process starts in the following cases:

- If a final node is found or all roles in $\mathscr{R}$ get synchronized. In that case, the *ProcessSeq* procedure is invoked to store in the *SEQS* array the current sequence of visited activities, as well as the causal poset corresponding to the sequence of visited posets.

- If a join node is found. This is done as part of the special treatment of fork-join pairs, as it is explained later on.

- In the presence of loops, when a third attempt to traverse the body of the loop is made. In this case the *ProcessSeq* procedure might or not be invoked. Consider the loop in Fig. 16(a). After visiting d_1 twice, *VisitSuccessorWC* may try to visit m_1 a third time with no success. It would then go back to d_1 and continue thereafter to v_3 without invoking *ProcessSeq*. Consider now the loop in Fig. 16(b). Here, after visiting v_2 twice, *VisitSuccessorWC* may try to visit m_1 a third time. As in the previous example, it would not succeed and the backtracking process would be initiated. However, before that, *VisistSuccessorWC* would invoke the *ProcessSeq* procedure (line 14). As a result, the activity sequence $v_1 \cdot v_2 \cdot v_2$ and the causal poset $ps_1 \circ_w ps_2 \circ_w ps_2$ would be stored in the *SEQS* array.

When a fork node is visited, the *MapForkWC* procedure is invoked (line 26). This procedure returns two sets, *SynchPaths* and *UnsynchPaths*, whose elements are tuples of the form $(seq, ps, \mathscr{R}')$ (i.e. the same type of tuples stored in the *SEQS* array). The elements in *SynchPaths* describe execution paths through the fork where all roles in $\mathscr{R}$ get synchronized. For each of these paths the *ProcessSeq* procedure is invoked (line 32) and the backtracking process is initiated. The elements in *UnsynchPaths* describe execution paths through the fork where not all roles in $\mathscr{R}$ get synchronized. For each of these paths, the *VisistSuccessorWC* procedure is invoked (line 37) to continue traversing the choreography graph from the fork's companion join node.

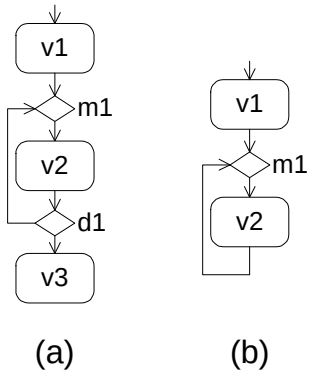

Figure 16: Two examples of loops in a choreography graph

Procedure *MapForkWC* (on page 53). For each branch of a fork-join pair, this procedure traverses all the possible execution paths, and generates a causal poset and a sequence of activities for each of them (lines 1-1). Thereafter, it groups the paths in sets *CP* of concurrent paths (the *getsComb* function, shown below, is used for this – line 9). For example, two sets of concurrent paths would be generated for the fork-join pair in Fig. 15, one for $v2 \cdot v3$ and $v_4 \cdot v_5$, and another for $v2 \cdot v3$ and $v_4 \cdot v_6$. The procedure generates a causal order for each set of concurrent paths, as well as a corresponding activity sequence, and stores them in the *SynchPaths* set, if all roles in $\mathscr{R}$ get synchronized in at least one of the concurrent paths, or in the *UnsynchPaths* set, otherwise.

$$getCombs(\mathbb{S}) \quad = \quad \begin{cases} \emptyset, \text{ if } \mathbb{S} = \emptyset \\ \{\{s\} : s \in S\}, \text{ if } \mathbb{S} = \{S\} \\ \{\{e\} \cup Comb : e \in S \wedge Comb \in getCombs(\mathbb{S}')\}, \text{ if } \mathbb{S} = \mathbb{S}' \cup \{S\} \end{cases}$$

Procedure *CheckRacesWC* (on page 54). This procedure analyzes the causal posets generated by the *VisitSuccessorWC* procedure (and stored in the global variable *SEQS*) in order to find race conditions. Each of those causal posets describes an execution path through a sequence of activities $v_1 \cdot v_2 \cdot \ldots \cdot v_n$, and is of the form $(E_\prec, \prec) = ps_1 \circ_w ps_2 \circ_w \ldots \circ_w ps_n$ $(n \geq 2)$, where $ps_1 \circ_w ps_2$ is weakly-causal.

CheckRacesWC checks whether any event $e \in E_1$ (if $ps_1 = (E_1, \prec_1)$) is in race with any receiving event event $r \in E_\prec - E_1$. This is done by checking whether e is a causal predecessor of r. If it is not, the events are in race, and the activity sequence where such race may happen is stored in the *EventsInRace* table. Note that the same poset may appear twice in $ps_1 \circ_w ps_2 \circ_w \ldots \circ_w ps_n$, if its activity is inside a loop in the choreography graph. In that case the second occurrence of the poset will have their events relabeled and marked as loop events. This fact is used by the *CheckRacesWC* procedure to determine the "type of loop race" (see explanation on page 41).

Algorithm 9: DetectRacesWithWeakCausality

Data: A choreography graph (V,E)
Result: x

foreach *pair* (v_1,v_2) *of activities whose sequential composition is weakly-causal* **do**
  /* $[\![SD]\!]^{\mathrm{WC}}_{\mathrm{SD}}$ instantiates each loop of SD with two iterations */
  foreach *poset* $ps_1 = (S_1 \cup R_1, \prec_1, t_1) \in [\![SD_{v_1}]\!]^{\mathrm{WC}}_{\mathrm{SD}}$ *of* v_1 **do**
    foreach *poset* $ps_2 \in [\![SD_{v_2}]\!]^{\mathrm{WC}}_{\mathrm{SD}}$ *of* v_2 *such that* $ps_1 \circ_w ps_2$ *is weakly causal* **do**
      /* Obtain the set $\mathscr{R}$ of roles that are subject to potential races due to weak causality */
      $\mathscr{R} \leftarrow \emptyset$; $MaxEv \leftarrow \emptyset$
      foreach *initiating-role* q *of* ps_2 **do**
        foreach *maximum event* $m \in max(ps_1)$ *such that* $loc(m) = q$ **do**
          if m *is a receiving event* **then** $m \leftarrow snd(m)$
          $MaxEv \leftarrow MaxEv \cup \{m\}$
          $\mathscr{R} \leftarrow \mathscr{R} \cup \{loc(s) \neq q : s \in S_1 \wedge (m \prec_1 s \vee (m \not\prec_1 s \wedge s \not\prec_1 m))\}$
      if $\mathscr{R} \neq \emptyset$ **then**
        /* Now we start a DFS on the graph, looking for activities where roles subject to races participate */
        forall $v \in V$ **do**
          if v *is an activity node* **then**
            $visited[ps_v] \leftarrow 0$, for each poset ps_v of v
          else
            $visited[v] \leftarrow 0$
        $forks \leftarrow 0$; $i \leftarrow 0$; $SEQS \leftarrow \emptyset$
        $AuxSeqAct \leftarrow \{v_1, v_2\}$; $AuxSeqPS \leftarrow \{ps_1, ps_2\}$
        $visited[ps_1]{+}{+}$; $visited[ps_2]{+}{+}$
        $seqProcessed \leftarrow false$
        $VisitSuccessorWC(u, \mathscr{R}, ps_2)$ // u is v_2's successor
        $CheckRacesWC(v_1, S_1 \cup R_1, MaxEv, \mathscr{R})$

Procedure `VisitSuccessorWC(`$v, \mathscr{R}, ps_{\text{prev}}$`)`

```
1  if v is an activity node then
2      R_bck ← R /* Backup to have fresh data for each poset                */
3      done ← false
4      foreach poset ps_v ∈ [[SD_v]]^WC_SD of v, such that visited[ps_v] < 2 do
5          R ← R_bck
6          visited[ps_v]++
           /* Check if any role gets "synch" with a sending event           */
7          R ← R − SynchronizedRoles(ps_prev, ps_v, R)
8          if R = ∅ ∧ !done then
9              done ← true
10             ProcessSeq(AuxSeqPS, AuxSeqAct, R)
11         else if R ≠ ∅ then
               /* NOTE: If visited[ps_v] = 2, relabel events of ps_v as loop events */
12             AuxSeqPS ← AuxSeqPS ∪ {ps_v}; AuxSeqAct ← AuxSeqAct ∪ {v}
13             VisitSuccessorWC(u, R, ps_v) // u is v's successor
14             if !seqProcessed then ProcessSeq(AuxSeqPS, AuxSeqAct, R)
15             AuxSeqPS ← AuxSeqPS − {ps_v}; AuxSeqAct ← AuxSeqAct − {v}
16         visited[ps_v]−−
17 else if visited[v] < 2 then
18     visited[v]++
19     if v is an activity-final or flow-final node then
20         ProcessSeq(AuxSeqPS, AuxSeqAct, R)
21     else if v is a join node AND forks > 0 then
22         ProcessSeq(AuxSeqPS, AuxSeqAct, R)
23         v_join ← v
24     else if v is a fork node then
25         forks++; v_join ← null
26         (SynchPaths, UnsynchPaths) ← MapForkWC(v, R, ps_prev)
27         forks−−
28         if v_join = null then
               /* Either all fork branches got synchronized before reaching the
               associated join node, or all branches ended up in a final node.
               In any case we are finished.                                  */
29             SynchPaths ← SynchPaths ∪ UnsynchPaths
30             UnsynchPaths ← ∅
31         foreach (forkActs, ps_fork, R_fork) ∈ SynchPaths do
32             ProcessSeq(AuxSeqPS ∪ {ps_fork}, AuxSeqAct ∪ {forkActs}, R_fork)
33         v_join_succ ← successor of v_join
34         foreach (forkActs, ps_fork, R_fork) ∈ UnsynchPaths do
35             AuxSeqPS ← AuxSeqPS ∪ {ps_fork}; AuxSeqAct ← AuxSeqAct ∪ {forkActs}
36             seqProcessed ← false
37             VisitSuccessorWC(v_join_succ, R_fork, ps_fork)
38     else
39         foreach u successor of v do
40             seqProcessed ← false
41             VisitSuccessorWC(u, R, ps_prev)
42     visited[v]−−
43 return
```

Procedure `ProcessSeq`($AuxSeqPS, AuxSeqAct, \mathscr{R}$)

1 $SEQS[i] \leftarrow (AuxSeqAct, CausalOrderSeq(AuxSeqPS), \mathscr{R})$
2 $i++$
3 $seqProcessed \leftarrow true$
4 **return**

Procedure `MapForkWC`($v_{fork}, \mathscr{R}, ps_{prev}$)

1 $AuxSeqPS_{\text{old}} \leftarrow AuxSeqPS$; $AuxSeqAct_{\text{old}} \leftarrow AuxSeqAct$; $i_{\text{old}} \leftarrow i$
2 $Paths \leftarrow \emptyset$;
3 **foreach** *successor* u *of* v_{fork} **do**
4 $AuxSeqPS \leftarrow \emptyset$; $AuxSeqAct \leftarrow \emptyset$; $i \leftarrow i_{\text{old}}$
5 $VisitSuccessorWC(u, \mathscr{R}, ps_{\text{prev}})$
`/* SEQS is a global variable updated by VisitSuccessorWC */`
6 Paths $\leftarrow$ Paths $\cup \{\{SEQS[j] : i_{old} \leq j < i\}\}$
7 $AuxSeqPS \leftarrow AuxSeqPS_{\text{old}}$; $AuxSeqAct \leftarrow AuxSeqAct_{\text{old}}$; $i \leftarrow i_{\text{old}}$
8 $SynchPaths \leftarrow \emptyset$; $UnsynchPaths \leftarrow \emptyset$
9 **foreach** $CP \in getCombs(Paths)$ **do**
10 $ps_{\text{fork}} \leftarrow CausalOrderPar(\{(ps : (seq, ps, \mathscr{R}_{\text{p}}) \in CP\})$
$forkActs \leftarrow \big|\big|_{(seq,ps,\mathscr{R}_{\text{p}}) \in CP} seq$
11
12 **if** $\exists (seq, ps, \mathscr{R}_p) \in CP$ *such that* $\mathscr{R}_p = \emptyset$ **then**
`/* If roles get synchronized in a path, they do it for the set of concurrent paths */`
13 $SynchPaths \leftarrow SynchPaths \cup \{(forkActs, ps_{\text{fork}}, \emptyset)\}$
14 **else**
15 $UnsynchPaths \leftarrow UnsynchPaths \cup \{(forkActs, ps_{\text{fork}}, \{\mathscr{R}_{\text{p}} : (seq, ps, \mathscr{R}_{\text{p}}) \in CP\})\}$
16 **return** $(SynchPaths, UnsynchPaths)$

Procedure `CheckRacesWC`$(v_1, E_1, \mathrm{MaxEv}, \mathscr{R})$

```
1  PMR_wc ← ∅; n ← 0
2  foreach j ∈ {0...i} do
3      (ActSeq, (E_≺, ≺), ℛ') ← SEQS[j]
4      R_tail ← {e ∈ (E_≺ − E_1) : e is a receiving event}
5      foreach p ∈ ℛ do
6          foreach e ∈ E_1, such that loc(e) = p do
7              foreach r ∈ R_tail, such that loc(r) = p do
8                  if e ⊀ r then
9                      if e or r are marked as a "loop event" then
                           /* e_type (resp. r_type) is the event type of which e
                           (resp. r) is an instance                              */
10                         (type1, type2) ← RaceType[e_type][r_type]
11                         if e is marked as a "loop event" then
12                             type1 ← true
13                         if r is marked as a "loop event" then
14                             type2 ← true
15                         RaceType[e_type][r_type] ← (type1, type2)
16                     else
17                         EventsInRace[e][r] ← EventsInRace[e][r] ∪ {SEQS[j].ActSeq}
18         PMRSeqs_wc[v_1][p][n] ← SEQS[j]
19         PMR_wc ← PMR_wc ∪ {(v_1, p, n)}
20         n++
```

5.1.3 Races in Chained Activity Sequences

Algorithm *DetectRacesInChainedActSeqs* (on page 57). Given an activity sequence $seq_1 = v_1 \cdot v_2 \cdot \ldots \cdot v_n$ $(n > 1)$, obtained by the *DetectRacesWithSendCausality* algorithm[19], the *DetectRacesInChainedActSeqs* algorithm looks for other activity sequences whose first activity is v_n, that is, sequences of the form $seq_2 = v_n \cdot v_{n+1} \cdot \ldots \cdot v_m$ $(m > n)$. Thereafter, it looks for races between v_1 and v_m, if seq_2 was obtained by the *DetectRacesWithSendCausality* algorithm. Otherwise, if seq_2 was obtained by the *DetectRacesWithWeakCausality* algorithm, it looks for races between v_1 and any of the activities in seq_2. In the former case the matching and detection processes are repeated again, now with $seq_1 = v_1 \cdot v_2 \cdot \ldots \cdot v_n \cdot v_{n+1} \cdot \ldots \cdot v_m$.

The matching and detection processes described above are indeed performed by the *CheckChainsSC* and *CheckChainsWC* procedures, which we explain in the following.

Procedure *CheckChainsSC* (on page 58). This procedure tries to chain the activity sequences that were obtained by the *DetectRacesWithSendCausality* algorithm (so it assumes that the sequential composition of activities is send-causal). When two or more sequences are chained, the procedure checks for races between the first and the last activities of the resulting sequence. The procedure also detects whether the activities in races are inside loops. If two activities v_1 and v_2, such that v_1 is executed before v_2[20], are in race, and v_1 is inside a loop, $LoopRaceType1[v_1][v_2]$ is set to *true*. This means that if an event e_1 from v_1 is in race with and event e_2 from v_2, all instances of e_1 (due to the loop iterations) will be in race with e_2. If v_2 is inside a loop, $LoopRaceType2[v_1][v_2]$ is set to *true*. This means that if an event e_1 from v_1 is in race with and event e_2 from v_2, e_1 will be in race with all instances of e_2. To detect the existence of loops, when an activity sequence is considered for concatenation, its last activity is marked as "visited", as well as added to the *visitedSeq* set, which is an ordered set of visited activities (line 8). Activities inside loops are stored in the *ActInsideLoop* set. In addition, the procedure uses the *ActInRace* set to keep record of the activities that are in race.

The procedure receives as input an activity sequence seq_1, which starts with an activity v and ends with an activity u. Using the indexes stored in PMR_{rcv} and PMR_{snd}, the procedure finds activity sequences that start with u. When a sequence seq_2 starting with u is found, its last activity (in the following w) is extracted. The procedure then checks whether w has already been visited (line 7). If not, it marks it as visited and checks whether v and w are the same activity. If that is the case, v is inside a loop, so it is added to the *ActInsideLoop* set. If any activity is in race with v the appropriate $LoopRaceType1$ flag is set to *true* (lines 9-11). After that, the procedure checks whether there are any races between events of v and

[19]Recall that in the *CheckRacesSC* procedure (pages 41 and 46), activity sequences (together with additional data) were stored in the $PMRSeqs_{\text{sc}}$ table, and that the indexes of the entries created in that table were stored in either the PMR_{rcv} set or the PMR_{snd} set. Activity sequences obtained by the *DetectRacesWithSendCausality* algorithm can thus be retrieved via the elements in PMR_{rcv} and PMR_{snd}.

[20]If the execution is tarted at the choreography graph's initial node

events of w (lines 12-15). If any race is detected, an entry in the *EvenstInRace* table is created, and w is added to the *ActInRace* set (line 16). The *LoopRaceType*1 flag is also set to *true* if v is inside a loop. Once the race detection process is finished, the procedure checks whether it is allowed to chain the activity sequence resulting from concatenating seq_1 and seq_2 with other activity sequences (line 18)[21]. If allowed, the procedure invokes either *CheckChainsSC* (i.e. a recursive invocation) or *CheckChainsWC* to continue with the chaining process. The former happens if role p does not execute any sending event in w. In that case, the index (u, p, j) pointing to seq_2 belongs to PMR_{rcv}.

If w was already visited, a loop has been detected. In that case all activities that were visited after the first visit to w are added to the *ActInsideLoop* set. Also, if any of those activities is in race with v, the appropriate *LoopRaceType*2 flag is set to *true*.

Procedure *CheckChainsWC* (on page 59). Given an activity sequence seq_1 (obtained by the *DetectRacesWithSendCausality* algorithm), which starts with an activity v and ends with an activity u, this procedure tries to find activity sequences obtained by the *DetectRacesWithSendCausality* algorithm that start with u (this is done using the indexes stored in PMR_{wc}). Given an activity sequence seq_2 that starts with u, and whose causal partial order is $(R_{\prec} \cup S_{\prec}, \prec)$, the procedure looks for races between any receiving event r_{v} from v (i.e. $r_{\text{v}} \in R^{v}_{\text{race}}$) and any receiving event r from seq_2 (i.e. $r \in R_{\prec}$). For that, the procedure obtains the set of minimum sending events of p in seq_2 (by construction of seq_2 those sending events should belong to u's set of events). Role p may have more than one minimum sending event if there are concurrent sendings. Given a maximum sending event m, and a receiving event $r \in R_{\prec}$, it is easy to see that if $m \prec r$, no $r_{\text{v}} \in R^{v}_{\text{race}}$ can be in race with r (since $r \prec m$ will always be true). Therefore, the procedure only looks for receiving events $r \in R_{\prec}$that have no maximum sending of p as causal predecessor.

[21] See explanation on page 38.

Algorithm 14: DetectRacesInChainedActSeqs

Data: $PMR, PMRSeqs$ from Algorithm 2; Choreography graph (V,E)
Result: A table *EventsInRace* containing in position (e_1,e_2) a set of collaboration sequences that lead to a race between events e_1 and e_2

foreach *role* p **do** $ActsInsideLoop[p] \leftarrow \emptyset$
forall $(v,p,i) \in PMR_{rcv}$ *such that* $v \neq synchPMR[v][p][i]$ **do**
  $(seq_1, R^v_{\text{race}}, R^u_{\text{race}}, race) \leftarrow PMRSeqs_{\text{sc}}[v][p][i]$
  $visitedSeq \leftarrow \emptyset$
  forall $u \in V$ **do** $visited[u] \leftarrow false$ `// Mark all choreography nodes as non-visisted`
  $x \leftarrow$last element of seq_1
  $visited[x] \leftarrow true$; $visitedSeq \leftarrow \{x\}$
  if $v = x$ **then**
    $ActsInsideLoop[p] \leftarrow ActsInsideLoop[p] \cup \{v\}$
  $ActsInRace \leftarrow \emptyset$
  if *race* **then** $ActsInRace \leftarrow \{x\}$
  $CheckChainsSC(seq_1, v, p, R^v_{\text{race}})$
forall $(v,p,i) \in PMR_{snd}$ *such that* $v \neq synchPMR[v][p][i]$ **do**
  $(seq_1, R^v_{\text{race}}, R^u_{\text{race}}, race) \leftarrow PMRSeqs_{\text{sc}}[v][p][i]$
  $u \leftarrow$last element of seq_1
  $CheckChainsWC(seq_1, u)$

Procedure `CheckChainsSC`$(seq_1, v, p, R^v_{race})$

```
1  visited_old ← visited; visitedSeq_old ← visitedSeq;
2  u ←last element of seq_1
3  forall (u,p,j) ∈ PMR_snd ∪ PMR_rcv, j ≥ 0 do
4      visited ← visited_old; visitedSeq ← visitedSeq_old;
5      (seq_2, R^u_race, R^w_race, race) ← PMRSeqs_sc[u][p][j]
6      w ←last element of seq_2
7      if !visited[w] then
8          visited[w] ← true; visitedSeq ← visitedSeq ∪ {w}
9          if w = v then
10             ActsInsideLoop[p] ← ActsInsideLoop[p] ∪ {v}
11             forall x ∈ ActsInRace do LoopRaceType1[v][x] ← true
12         foreach r_v ∈ R^v_race do
13             foreach r_w ∈ R^w_race do
14                 if non-FIFO channels OR (FIFO channels AND
                   loc(snd(r_v)) ≠ loc(snd(r_w))) then
15                     EventsInRace[r_v][r_w] ← EventsInRace[r_v][r_w] ∪ {(seq_1 − {u}) ∪ seq_2}
16                     ActInRace ← ActInRace ∪ {w}
17                     if v ∈ ActInsideLoop[p] then LoopRaceType1[v][w] ← true
18         if synchPMR[u][p][j] = null ∨ (synchPMR[u][p][j] ≠ v ∧ !visited[synchPMR[u][p][j]])
           then
19             if (u,p,j) ∈ PMR_rcv then
20                 CheckChainsSC((seq_1 − {u}) ∪ seq_2, v, p, R^v_race)
21             else
22                 CheckChainsWC((seq_1 − {u}) ∪ seq_2, p, R^v_race)
23     else
24         X ← {w} ∪ {x : x appears after w in visitedSeq}
25         ActsInsideLoop[p] ← ActsInsideLoop[p] ∪ X
26         forall x ∈ ActsInRace ∩ X do LoopRaceType2[v][x] ← true
27 return
```

Procedure `CheckChainsWC`(seq_1, p, R^v_{race})

```
1  u ←last element of seq_1
2  forall (u, p, j) ∈ PMR_wc, j ≥ 0 do
3      (seq_2, (R_≺ ∪ S_≺, ≺), ℛ) ← PMRSeqs_wc[u][p][j]
4      M ← {m : m ∈ min((S_≺, ≺)) ∧ loc(m) = p}  // Minimum sending event(s) of p in E_≺
5      foreach r_v ∈ R^v_race do
6          foreach r ∈ R_≺ such that loc(r) = p ∧ m ⊀ r, ∀m ∈ M do
7              if non-FIFO channels OR (FIFO channels AND loc(snd(r_v)) ≠ loc(snd(r)))
               then
8                  if r is marked as a "loop event" then
                       /* We assume r_type is the event type of which r is an
                       instance                                              */
9                      (type1, type2) ← RaceType[r_v][r_type]
10                     type2 ← true
11                     if v ∈ ActsInsideLoop[p] then type1 ← true else type1 ← false
12                     RaceType[r_v][r_type] ← (type1, type2)
13                 else
14                     EventsInRace[r_v][r] ← EventsInRace[r_v][r] ∪ {(seq_1 − {u}) ∪ seq_2}
15 return
```

5.2 Detection of Ambiguous and Race Propagation

Ambiguous propagation happens when the triggering traces *specified* for a non-choosing component in two different alternatives of a choice have a common prefix. Race propagation happens if there can be an ambiguous propagation due to the existence of races. That is, if the triggering traces that may be *observed* at run-time in two different alternatives, due to the effect of races, have a common prefix.

Obviously, in order to determine whether two triggering traces have a common prefix it is sufficient to compare their first elements. This is indeed enough to either assert or negate the existence of ambiguous or race propagation. However, in case of a propagation problem, knowing the complete triggering traces and their complete common prefix helps to determine the most appropriate resolution. For example, consider these two pairs of triggering traces, both of them giving rise to ambiguous propagation: $\{(?a),(?a)\}$ and $\{(?a,?b,?c),(?a,?b,?d)\}$. Whit the second pair we may opt for a design solution where the decision on which alternative to follow is postpone until the reception of either message c or d. We can do this by "extracting" $?a,?b$ from the choice in the local behavior of the non-choosing component. Obviously, this solution is not valid for the first pair of triggering traces.

We present in the following an algorithm for the detection of both ambiguous and race propagation problems. In the case of ambiguous propagation, the algorithm is able to obtain the maximum common prefix of two triggering traces leading to ambiguous propagation, even in the presence of loops. In the case of race propagation, the algorithm considers only one iteration of loops. This means that race propagation problems will always be detected, but the maximum common prefix of the triggering traces leading to a race propagation problem may not always be obtained.

For the algorithms we assume that a single state machine will be synthesized for each role in the choreography graph, and that such state machine will have one single input buffer to store the messages received from all the other roles. In the case that the same role participates in two concurrent collaborations (i.e. inside a fork-join pair), we consider that the messages received and sent by the role in both collaborations are interleaved (since we consider one single state machine with one single buffer for the role). However, a decision could be made to create the role state machine in such a way that messages belonging to each of the concurrent collaborations are treated by separate orthogonal sub-state machines. In that case, messages would no longer interleave. The proposed algorithms could be easily adapted to consider this case. We just need to mark the roles in concurrent collaborations as implemented by different state-machines. A decision could also be made to provide state machines with different input buffers for different peer roles. In that case, to construct correct triggering traces, the algorithm should be modified to consider only messages that can be received on the same buffer.

Algorithm *DetectChoicePropagationProblems* (on page 69). For each choice node v_{ch}, and each non-choosing role p on v_{ch}, this algorithm invokes the

ChoreographyToFSA procedure to convert (a part of) the choreography graph into an "equivalent" (from the point of view of p) finite state automaton, whose transitions are labeled by receiving and sending events executed by p. The resulting automaton may also have ε-transitions (i.e. silent transitions). Those ε-transitions are removed (see Appendix B.2), and the resulting automaton is split into as many automata as branches has the choice under analysis. Each of the new automata may be used to generate the triggering traces of one of the choice's branches.

These "branch" automata are then passed as input to the *DetectAmbiguousPropagation* and *DetectRacePropagation* procedures, which will analyse them for the detection of ambiguous and race propagation problems.

Procedure *DetectAmbiguousPropagation* (on page 69). This procedure receives as input a set $\mathbb{A}$ of automata, where each automaton $\mathscr{A}_i \in \mathbb{A}$ describes (part of) the behavior of a non-choosing role p in i^{th} branch of the choice under analysis. More specifically, $\mathscr{A}_i$ describes the behavior of p, up to the its first sending event (if any), in each of the possible execution paths through the i^{th} branch. That is, $\mathscr{A}_i$ describes p's *specified* triggering traces in the i^{th} branch of the choice under analysis.

Given two automata $\mathscr{A}_1 = (Q_1, \Sigma_1, \delta_1, q_{01}, F_1)$ and $\mathscr{A}_2 = (Q_2, \Sigma_2, \delta_2, q_{02}, F_2)$, an ambiguous propagation will exist if one of the strings accepted by $\mathscr{A}_1$ has a common prefix with one of the strings accepted by $\mathscr{A}_2$. This can be easily checked by constructing their intersection automaton $(Q, \Sigma, \delta, q_0, F) = \mathscr{A}_1 \cap \mathscr{A}_2$, where $Q = Q_1 \times Q_2$, $\Sigma = \Sigma_1 \cap \Sigma_2$, $\delta = \{((q_1, q_2), e, (q_1', q_2')) : (q_1, e, q_1') \in \delta_1, (q_2, e, q_2') \in \delta_2\}$, $q_0 = (q_{01}, q_{02})$ and $F = F_1 \times F_2$.

Note that before intersecting the automata, their transitions are relabeled, so that instead of events, their associated messages (i.e. the message sent or received) are used as transition labels.

If the set of transitions of the intersection automaton is non-empty (i.e. $\delta \neq \emptyset$), there is a problem of ambiguous propagation. To obtain the sequence(s) of events leading to ambiguous propagation, the intersection automaton is converted into an equivalent regular expression. Note, however, that the intersection automaton may have unreachable states. These states should be eliminated before converting the automaton into a regular expression. In addition, ε-transitions should be added from each state with no output transitions to a common final state.

Procedure *DetectRacePropagation* (on page 70). This procedure receives as input a set $\mathbb{A}$ of automata, where each automaton $\mathscr{A}_i \in \mathbb{A}$ describes (part of) the behavior of a non-choosing role p in i^{th} branch of the choice under analysis. More specifically, $\mathscr{A}_i$ describes the behavior of p, up to the its first sending event, in each of the possible execution paths through the i^{th} branch. That is, $\mathscr{A}_i$ describes p's *specified* triggering traces in the i^{th} branch of the choice under analysis. The procedure also gets as input the *containerNode* table, which for each sending event at the end of a triggering trace stores the activity node where the sending event can be found, or the fork node of a fork-join pair containing the activity where the sending event can be found.

To detect a race propagation this procedure obtains the *observed* triggering traces (lines 1-18), that is, the triggering traces observed by p in the presence of races. Once they have been obtained, the procedure checks whether any two of those traces (corresponding to different branches of the choice under analysis) have a common prefix (line 19).

To obtain the observed triggering traces the procedure proceeds as follows. Each automaton is converted into an equivalent regular expression (line 3), following the technique described in Appendix 84. The resulting regular expression may describe loops. The *SeparateInAltSubexpressions* procedure (line 3) initializes those loops, so at most one iteration is considered. For that purpose, each sub-expression $\alpha_1 \cdot \alpha_2^* \cdot \alpha_3$ is replaced with $(\alpha_1 \cdot \alpha_3 | \alpha_1 \cdot \alpha_2 \cdot \alpha_3)$, and each sub-expression α^+ is replaced with α. The new regular expression will consist of the union of several sub-expressions, each of them describing p's triggering trace in a given execution path. Each of those sub-expression will be processed separately. For each sub-expression re, the terminating sending event s, if any, is extracted. All events in re (except s) are then partially ordered (line 6): events in race become unordered; otherwise the total order dictated by re is respected. If re ended with a receiving event (i.e. $s = null$), it means that a final node was reached in the execution path described by re. The resulting poset can thus already be used to generate the observed triggering traces, which correspond to the labeled-linearizations of the poset[22] (line 18). Otherwise, if re ended with a sending event s, races might be possible between receiving events from the triggering trace and receiving events that succeed s according to the visual order. The set E_{ext} of "external" events (i.e. not from the triggering trace) that are in race with events of the triggering trace can be obtained with help of the *EventsInRace* table (lines 11-13). However, we still need to determine the causal order between those events, and between them and the events of the triggering trace. This is done with help of the *GetPosetsForObservedTT* procedure, which returns a set of partial orders whose labeled-linearizations correspond to the observed triggering traces.

Procedure *ChoreographyToFSA* (on page 71). This procedures returns an automaton describing the behavior of role p in (part of) the choreography graph. It also returns a table *containerNode*, which which stores the choreography nodes where some events can be found.

Starting from a decision node, this procedure traverses the choreography graph using a depth-first search technique [AHU74]. For each possible execution path starting at the decision node, the procedure stops searching when an activity is found where role p executes a sending event, or when a final node is reached. The nodes that are visited and the edges that are traversed are mapped into states and transitions of an automaton $\mathscr{A}$.

When an activity v where p participates is visited, function *fsa* is invoked to obtain an automaton describing the behavior of p in v (line 4). This new automaton

[22] A *linearization* of a poset $(E, <)$ is a word $w = e_1 \cdot \cdots \cdot e_{|E|}$ over the alphabet E, such that if $e_i < e_j$ then $i < j$. In [ON05] a technique is described to obtain all possible linearizations of a poset in an efficient way. A *labeled-linearization* is a linearization where each event has been replaced with its associated message (i.e. the message sent or received).

is then concatenated with $\mathscr{A}$ at certain *junction* states J (i.e. final states with incoming transitions that are labeled with a receiving event – see *Concatenation* on page 65), and the set of junction states is updated. If the resulting automaton has any final state with an input transition labeled with a sending event s, an entry in the *containerNode* table is created. Thereafter, if all the final states of the automaton have input transitions labeled with sending events (i.e. the new set of junction states is empty – see line refjunctionempty), the backtracking process is initiated. Otherwise, the graph traversal continues.

When a merge node is found (line 9), a new state is added to $\mathscr{A}$ and each of the junction states is connected to the newly added state by means of ε-transitions. Decision nodes are not explicit mapped into states of $\mathscr{A}$. Instead, the junction states at the time a decision node is visited will act as "decision" states (i.e. the automata obtained by traversing each of the branches of the decision node will be concatenated at those junction states).

When a fork node is visited (line 11) the *ForkToFSA* procedure is invoked. This procedure returns an automaton corresponding to the nodes found within the fork and its companion join node. This automaton is then concatenated with $\mathscr{A}$. If the set of new junction states is empty (i.e. a sending event was found in all paths within the fork-join pair) or if a join node was not found (because all the fork's branches ended up in a final node), the backtracking process is initiated. Otherwise, the graph traversal continues from the fork's companion join node.

Each node in the choreography graph is visited at most once. It may happen that an attempt is made to re-visit a merge node, or to re-visit the decision node of the choice under analysis (i.e. the very first node been visited). The latter may happen if that node is inside a loop, as in the choreography of Fig. 17(a). In the first case ε-transitions are added from the current junction states to the state corresponding to the merge node (i.e. the state added the first time the merge node was visited)(line 28). In the second case , ε-transitions are added from the current junction states to the initial state (line 30).

Figure 17 shows a choreography graph and the automaton created by procedure *ChoreographyToFSA* for role *R2*.

Function *fsa*. This function takes an expression describing a sequence diagram[23] and a role p, and returns an automaton $\mathscr{A}$ describing the behavior of p in the given sequence diagram. It is defined as

$$fsa(SD,p) = \begin{cases} \mathscr{A}_{bSD}, \text{ if } SD := bSD \\ fsa(SD_1,p) \cdot fsa(SD_2,p), \text{ if } SD := SD_1\, seq\, SD_2 \\ fsa(SD_1,p) \cup fsa(SD_2,p), \text{ if } SD := SD_1\, alt\, SD_2 \\ fsa(SD_1,p) \times fsa(SD_2,p), \text{ if } SD := SD_1\, par\, SD_2 \\ fsa(SD_1,p)^*, \text{ if } SD := loop(0,n)\, SD_1, n > 0 \\ fsa(SD_1,p) \cdot fsa(SD_1,p)^*, \text{ if } SD := loop(n,m)\, SD_1, 0 < n \leq m \end{cases}$$

[23]Recall that a sequence diagram can be described by an expression consisting of basic sequence diagrams (bSDs) identifiers and a combination of `seq`, `alt`, `par` and `loop` operators (i.e. an expression conforming to the BNF-grammar described in Sect. 5)

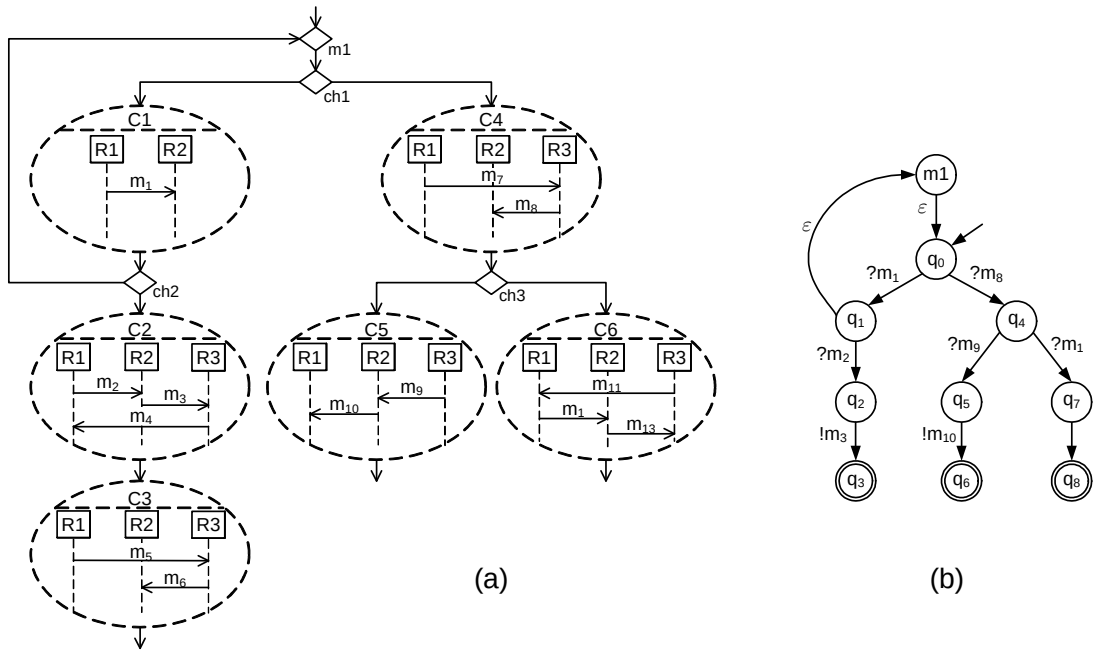

Figure 17: (a) Choice (ch1) with race propagation; (b) Automaton describing the significant part of *R2*'s behavior in (a) for detection of race propagation

where $\mathscr{A}_{bSD}$ is an automaton describing the behavior of p, up to its first sending event (if any), in the *bSD* basic sequence diagram. To obtain $\mathscr{A}_{bSD}$, the basic sequence diagram is first projected onto the lifeline of p. The result is a totally ordered set of events $(E,<)$. From that set we are only interested in the first sending event (if any) and all its preceding events. That is, assuming that $E = R \cup S$ (with S the set of sending events), we are interested on the totally ordered set $(E',<)$, where

$$E' = \begin{cases} E, \text{ if } S = \emptyset \\ \{e \in E : (e < s \vee e = s), s \in S \wedge \nexists s' \in S, s' < s\}, \text{ otherwise} \end{cases}$$

The ordered set $(E',<)$ is then converted into an automaton $\mathscr{A}_{bSD}$ such that:

- If $E' = \emptyset$ (i.e. p does not participate in the basic sequence diagram), $\mathscr{A}_{\text{bSD}} = \mathscr{A}_{\emptyset}$, where $\mathscr{A}_{\emptyset} = (\{q_0\}, \emptyset, \emptyset, q_0, \{q_0\})$ is the so-called *empty* automaton.
- Otherwise, $\mathscr{A}_{\text{bSD}} = (Q, E, \delta, q_0, F)$, with

  $Q = \{q_0\} \cup \{q_e : e \in E\}$

  $\delta = \{(q_0, e, q_e) : e \in E \wedge \nexists f \in E, f < e\} \cup \{(q_e, f, q_f) : e, f \in E, e < f \wedge \exists g \in E, e < g < f\}$

  $F = \{q_e : e \in E \wedge \nexists f \in E, e < f\}$

The operators used by the *fsa* function to compose the automata are defined as follows. Let $\mathscr{A}_1 = (Q_1, \Sigma_1, \delta_1, q_{01}, F_1)$ and $\mathscr{A}_2 = (Q_2, \Sigma_2, \delta_2, q_{02}, F_2)$ be two automata with disjoint sets of states, and let $J_i = \{q_f \in F_i : \exists (q, r, q_f) \in \delta_i$ and r is a receiving event$\}$

, $i \in \{1,2\}$, be the set of *junction* states of $\mathscr{A}_i$ (i.e. the set of final states that have an incoming transition labeled with a receiving event).We then define the following four basic operations on automata:

Concatenation. For the concatenation of two automata $\mathscr{A}_1$ and $\mathscr{A}_2$, for each output transition with label e from the initial state of $\mathscr{A}_2$, new transitions are added, with the same label e, from the junction states of $\mathscr{A}_1$ to the successors of the initial state of $\mathscr{A}_2$. If the initial state of $\mathscr{A}_2$ is not a final state, it is removed, together with its output transitions. More formally, the concatenation $\mathscr{A}_1 \cdot \mathscr{A}_2$ of two automata $\mathscr{A}_1$ and $\mathscr{A}_2$ is an automaton $\mathscr{A}$ such that:

- $\mathscr{A} = \mathscr{A}_1$, if $J_1 = \emptyset \vee \mathscr{A}_2 = \mathscr{A}_\emptyset$
- $\mathscr{A} = \mathscr{A}_2$, if $\mathscr{A}_1 = \mathscr{A}_\emptyset$
- Otherwise, $\mathscr{A} = (Q, \Sigma_1 \cup \Sigma_2, \delta, q_{01}, F_2)$, with

  $$Q = \begin{cases} Q_1 \cup Q_2, \text{ if } q_{02} \in F_2 \\ Q_1 \cup Q_2 - \{q_{02}\}, \text{ otherwise} \end{cases}$$

  $$\delta = \delta_1 \cup (\delta_2 \cap (Q \times \Sigma_2 \times Q)) \cup \{(q_1, e, q_2) \in J_1 \times \Sigma_2 \times Q_2 : (q_{02}, e, q_2) \in \delta_2\}$$

Union. For the union of two automata, a new initial state is created and concatenated with each of the automata. More formally, the union $\mathscr{A}_1 \cup \mathscr{A}_2$ of two automata $\mathscr{A}_1$ and $\mathscr{A}_2$ is automaton $\mathscr{A}$, such that:

- $\mathscr{A} = \mathscr{A}_1$, if $\mathscr{A}_2 = \mathscr{A}_\emptyset$
- $\mathscr{A} = \mathscr{A}_2$, if $\mathscr{A}_1 = \mathscr{A}_\emptyset$
- Otherwise, $\mathscr{A} = (Q, \Sigma_1 \cup \Sigma_2, \delta, q_0, F_1 \cup F_2)$, where

  q_0 is a new state

  $$Q = \begin{cases} Q_1 \cup Q_2, \text{ if } q_{01} \in F_1 \text{ and } q_{02} \in F_2 \\ Q_1 \cup Q_2 - \{q_{01}, q_{02}\}, \text{ if } q_{01} \notin F_1 \text{ and } q_{02} \notin F_2 \\ Q_1 \cup Q_2 - \{q_{0i}\} \text{ if } q_{0i} \in F_i, i \in \{1,2\} \end{cases}$$

  $$\delta = (\delta_1 \cap (Q \times \Sigma_1 \times Q)) \cup (\delta_2 \cap (Q \times \Sigma_2 \times Q)) \cup \{(q_0, e, q_1) \in \{q_0\} \times \Sigma_1 \times Q_1 : (q_{01}, e, q_1) \in \delta_1\} \cup \{(q_0, e, q_2) \in \{q_0\} \times \Sigma_2 \times Q_2 : (q_{02}, e, q_2) \in \delta_2\}$$

Kleene closure. For the Kleene closure of an automaton, the initial state is also made a final state. The original final states are removed, and their input transitions are connected to the initial state. More formally, the Kleene closure $\mathscr{A}_1^*$ of an automaton $\mathscr{A}_1$ is an automaton $\mathscr{A}$ such that:

- $\mathscr{A} = \mathscr{A}_1$, if $F_1 = J_1 \vee \mathscr{A}_1 = \mathscr{A}_\emptyset$

- Otherwise, $\mathscr{A} = (Q, \Sigma_1, \delta, q_{01}, F)$, with

  $Q = Q_1 - J_1 \cup \{q_{01}\}$

  $\delta = \delta_1 \cup \{(q, e, q_{01}) : (q, e, q_j) \in \delta_1, q_j \in J_1\}$

  $F = \{q_{01}\} \cup (F_1 - J_1)$

Cross-product. The cross-product $\mathscr{A}_1 \times \mathscr{A}_2$ of two automata $\mathscr{A}_1$ and $\mathscr{A}_2$ is the automaton $\mathscr{A}$ returned by the *CrossProductFSA* procedure. That is, $\mathscr{A} = \mathit{CrossProductFSA}(\mathscr{A}_1, \mathscr{A}_2)$. The result is an automaton describing the interleaving of the transitions of the original automata. The interleaving is however stopped as soon as a transition labeled with a sending event is found.

Procedure *ForkToFSA* (on page 72). This procedure returns an automaton describing the behavior of a role p on the part of a choreography graph included between a fork node and its companion join node. It also returns a *containerNode* table, which for each sending event labeling an input-transition of a final automaton state (i.e. a sending event at the end of a triggering trace) stores the activity node where the sending event can be found, or the fork node of a fork-join pair containing the activity where the sending event can be found.

The procedure converts each branch of the fork into an automaton with help of the *ChoreographyToFSA* procedure. The resulting automaton may has several final states if several execution paths are possible through the fork's branch. In that case the *SplitFSA* procedure is invoked to split the automaton into as many automata as final states had the former automaton. Each of the new automata will describe the behavior of p on one of the execution paths through the branch. If any of the automata has a final state with input-transitions labeled with a sending event, an entry in the *containerNode* table is created storing the fork node under analysis.

After all branches have been processed, their associated automata are grouped in sets of concurrent automata (i.e. automata corresponding to concurrent execution paths) with help of the *getsComb* function (see page 50 for details on this function). A cross-product automaton is then obtained for each group of concurrent automata. The union of all the cross-product automata, and the *containerNode* table, are returned by the *ForkToFSA* procedure.

Procedure *GetPosetsForObservedTT* (on page 74). For a given role p, this procedure takes a poset $(E_{\mathrm{tt}}, \prec_{\mathrm{tt}})$, describing the causal order between the events of a certain triggering trace, a set E_{ext} of all "external" events (i.e. not from the triggering trace) that are in race with events from the triggering trace. Those external events may not all be executed in the same execution path. For each possible execution path, this procedure finds out which external events are executed in the given path and creates a causal poset (in the following *OTT poset*) relating those events and the events from the triggering trace. The set of all OTT posets is then returned.

In order to properly group the external events and create the observed triggering trace's posets, a choreography node v_0 is also provided as input for the procedure.

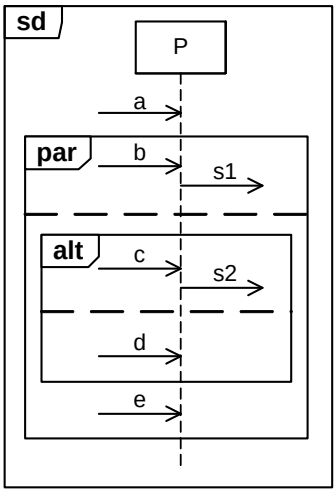

Figure 18: Behavior of role p in v_0

This node might be either an activity node or a fork node. Let us first look at the former case.

If v_0 is an activity node, v_0 is the activity where the sending event triggered by the triggering trace can be found. Since all external events will be successor (or concurrent) events of that sending event, the procedure uses v_0 as a starting point to traverse the choreography graph in search of the external events. Since the behavior of v_0 may be described by more than one poset, the procedure first needs to determine which of those posets should be considered as the actual starting point to begin searching for external events. The set PS_{match} of selected posets will contain those posets with the highest number of events in common with the triggering trace (line 3). Consider, for example, that the behavior of p in v_0 is the one illustrated in Fig. 18, and that $E_{\text{tt}} = \{\ldots, ?a, ?b, ?c\}$ (i.e. the triggering trace is, for example, $\ldots \cdot ?a \cdot ?c \cdot ?b$ and triggers $!s_1$). Then, of the two posets describing the behavior of v_0 the procedure will only select the poset that includes events $?c$ and $!s_2$.

Once the set PS_{match} has been obtained, the procedure determines the subset of external events that may be found in activities succeeding v_0 (line 5). If that subset is not empty, the *VisitSuccessorRP* procedure is invoked (line 10). That procedure traverses the choreography graph from v_0's successor using a DFS technique. If an activity where external events can be found is visited by *VisitSuccessorRP*, those external events are added into a poset. *VisitSuccessorRP* returns a set PS_{tail} of posets, each of them describing the causal order of the external events found on a certain execution path.

After *VisitSuccessorRP* returns (if it was invoked), the procedure analyses the poset in poset PS_{match}. For each poset $ps_0 \in PS_{\text{match}}$, the procedure checks whether any external event can be found in ps_0 (line 13). At this point it is important to note that not all external events that could be found in ps_0 should be included in the OTT poset under construction. Consider again the example in Fig. 18, and imagine

that $?a$ is in race with $?e$, but that this race can only happen when $?d$ is executed. Such race could therefore not happen for the given triggering trace. Although event $?a$ would be part of E_{ext}, it should not be included in the OTT poset. In general, for a poset ps_0, only external events that are not causal successor of any minimum sending of ps_0 should be included in the OTT poset. To determine the causal order between such events and the triggering trace events, two cases are considered. For those triggering trace events that can also be found in ps_0, the causal order relations dictated by that poset are considered (line 14). For each triggering trace event r_1 preceding the events of ps_0, a causal order relation $r_1 \prec_{\text{ott}} r_2$ is created for each selected external event r_2, if r_1 and r_2 are not in race (line 15).

Once ps_0 has been processed, the procedure checks whether role p gets synchronized on a sending event belonging to ps_0 (line 16). If p does not gets synchronized (line 5 of *UpdatePoset* procedure), each of the posets in PS_{tail} are concatenated with the OTT poset. If p gets synchronized, the triggering trace events from activities preceding v_0 will not be in race with events from activities succeeding v_0. However, triggering trace events from v_0 might be in race with events from activities succeeding v_0. If there is any triggering trace event from v_0 in race with some external events E (line 9), only the parts of the posets in PS_{tail} that deal with any of the E events are concatenated with the OTT poset. If $E = \emptyset$, the PS_{tail} posets are not considered to create the OTT posets.

As we already said, v_0 might also be a fork node. In that case, v_0 is the fork of a fork-join pair that contains the activity where the sending event triggered by the triggering trace can be found, but that does not contain all the events of the triggering trace. In this case, procedure *MapForkWC* (see page 50) is invoked to obtain a set of causal posets for all the activities contained by the fork-join pair. After that, the processing is similar as the case where v_0 is an activity.

Algorithm 17: DetectChoicePropagationProblems

```
1  foreach choice node v_ch ∈ V do
2      foreach non-choosing role p in v_ch do
3          forall v ∈ V do visited[v] = false
4          q_0 ← NewFSAState()
5          𝒜 ← ({q_0}, ∅, ∅, q_0, {q_0})
6          (𝒜, containerNode) ← ChoreographyToFSA(p, v_ch, 𝒜, {q_0})
7          𝒜 ← RemoveEpsilonTransitions(𝒜)
           /* We split 𝒜 in a set of automata, one for each branch of the
           choice.                                                        */
8          forall q ∈ Q such that ∃(q_0, e, q) ∈ δ, e ∈ Σ do
9              Q_cl ← Closure(q) /* All states reachable from q, incl. q */
10             F' ← Q_cl ∩ F; Q' ← Q_cl ∪ {q_0}; δ' ← {(q_1, e, q_2) ∈ δ : q_1, q_2 ∈ Q'}
11             𝔸 ← 𝔸 ∪ {(Q', Σ, δ', q_0, F')}
12         DetectAmbiguousPropagation(𝔸)
13         DetectRacePropagation(p, containerNode, 𝔸)
14 end
```

Procedure `DetectAmbiguousPropagation(𝔸)`

```
1  RE ← ∅
2  foreach 𝒜 ∈ 𝔸 do 𝒜 ← RelabelTransitions(𝒜)
3  foreach 𝒜_1 ∈ 𝔸 do
4      𝔸 ← 𝔸 − {𝒜_1}
5      foreach 𝒜_2 ∈ 𝔸 do
6          𝒜_prefix ← 𝒜_1 ∩ 𝒜_2 // Intersection automaton
7          if 𝒜_prefix.δ ≠ ∅ then
               /* Eliminate non-reachable states in 𝒜_prefix and add ε-transition
               from states w/o output transitions to a common final state  */
8              RE ← RE ∪ {ConvertFSAtoRE(𝒜_prefix)}
```

Procedure `DetectRacePropagation`$(p, containerNode, \mathbb{A})$

```
1  foreach 𝒜 ∈ 𝔸 do
2      ℒ[𝒜] ← ∅
3      RE ← SeparateInAltSubexpressions(ConvertFSAtoRE(𝒜))
4      foreach re ∈ RE do
5          s ← sending event at the end of re or null if re ends with a receiving event
6          (E_tt, ≺_tt) ←events in re (except s) are partially ordered
7          if s = null then
8              ϒ ← {(E_tt, ≺_tt)}
9          else
               /* Get set E_ext of "external" events in race with events from the
               triggering trace                                                  */
10             E_ext ← ∅
11             foreach e1 ∈ E_tt do
12                 foreach e2 such that EventsInRace[e1, e2] ≠ ∅ ∧ e2 ∉ E_tt do
13                     E_ext ← E_ext ∪ {e2}
14             if E_ext = ∅ then
15                 ϒ ← {(E_tt, ≺_tt)}
16             else
                   /* Get set ϒ of causal posets describing causal relations
                   between events in E_tt ∪ E_ext                                */
17                 ϒ ← GetPosetsForObservedTT(p, containerNode[s], E_ext, E_tt, ≺_tt)
18     ℒ[𝒜] ← ℒ[𝒜] ∪ GetLinearizations(ϒ)
19 if ∃l ∈ ℒ[𝒜1], 𝒜1 ∈ 𝔸 ∧ ∃l′ ∈ ℒ[𝒜2], 𝒜2 ∈ 𝔸 such that l and l′ have a common prefix then
20     Report race propagation
```

Procedure `ChoreographyToFSA(`$p, v, \mathscr{A}, J$`)`

```
1  containerNode ← ∅
2  visited[v] ← true
3  if v is an activity node where p participates then
       /* Let SD_v be the sequence diagram describing v's behavior */
4      (𝒜, J) ← ConcatenateFSA(𝒜, fsa(SD_v, p), J)
5      foreach sending event s such that (q, s, q_f) ∈ δ, q_f ∈ F − J do
6          containerNode[s] ← v
7      if J = ∅ then
8          visited[v] ← false; return (𝒜, containerNode)
9  else if v is a merge node then
10     q_v ← NewFSAState(v); Q ← Q ∪ {q_v}; δ ← δ ∪ {(q_j, ε, q_v) : q_j ∈ J}; F ← F ∪ {q_v}; J ← {q_v}
11 else if v is a join node then
12     v_join ← v; visited[v] ← false; return (𝒜, containerNode)
13 else if v is a fork node then
14     v_join ← null
15     (𝒜_fork, containerNode_aux) ← ForkToFSA(p, v)
16     containerNode ← containerNode ∪ containerNode_aux
17     (𝒜, J) ← ConcatenateFSA(𝒜, 𝒜_fork, J)
18     if J = ∅ ∨ v_join = null then
19         visited[v] ← false
20         return (𝒜, containerNode)
       /* We continue traversing the graph from the join node associated to the
       fork (i.e. v_join). Note that we assume proper nesting of fork/join
       nodes */
21     v ← v_join
22 foreach u successor of v do
23     if !visited[u] then
24         (𝒜, containerNode_aux) ← ChoreographyToFSA(p, u, 𝒜, J)
25         containerNode ← containerNode ∪ containerNode_aux
26     else
27         if u is a merge node then
28             δ ← δ ∪ {(q_j, ε, q_u) : q_j ∈ J}
29         else
30             δ ← δ ∪ {(q_j, e, q_0) : q_j ∈ J}
31 if v is NOT a merge node then
32     visited[v] ← false
33 return (𝒜, containerNode)
```

Procedure `ForkToFSA(`p, v_{fork}`)`

```
1  A_all ← ∅
2  foreach successor v of v_fork do
       /* An FSA is built for each branch of the fork                    */
3      q_0 ← NewFSAState(v)
4      A_v ← ({q_0}, ∅, ∅, q_0, {q_0})
5      (A_v, containerNode) ← ChoreographyToFSA(p, v, A_v, {q_0})
6      A_v ← RemoveEpsilonTransitions(A_v)
       /* If A_v has several final states, SplitFSA() returns one FSA for each
       final state                                                       */
7      if |A_v.F| > 1 then
8          A_all ← A_all ∪ {SplitFSA(A_v)}
9      else
10         A_all ← A_all ∪ {{A_v}}
11     foreach s such that (q, s, q_f) ∈ A_v.δ, q_f ∈ A_v.F and s ∈ A_v.Σ_v is a sending event} do
12         containerNode[s] ← v_fork
13 A_fork ← ({q_0}, ∅, ∅, q_0, {q_0})
14 foreach A ∈ getCombs(A_all) do
15     A_fork ← A_fork ∪ CrossProductFSA(A)
16 return (A_fork, containerNode)
```

Procedure `CrossProductFSA(`$\mathbb{A}$`)`

```
   Data: Set A of automata
   Result: FSA (Q, Σ, δ, q_0, F) corresponding to the cross-product of the FSAs in A
1  if A = {A} then return A
2  A_1 ← any element of A; A ← A − {A_1}
3  while A ≠ ∅ do
4      A_2 ← any element of A; A ← A − {A_2}
5      Q_aux ← {(q_01, q_02)}
6      while Q_aux ≠ ∅ do
7          (q_1^s, q_2^s) ←Any element of Q_aux; Q_aux ← Q_aux − {(q_1^s, q_2^s)}
8          foreach transition (q_1^s, e_1, q_1^d) ∈ δ_1 do
9              q_new ← (q_1^d, q_2^s); δ ← δ ∪ {((q_1^s, q_2^s), e_1, q_new)}; Q ← Q ∪ {q_new}
10             if e_1 is a sending event ∨ q_new ∈ F_1 x F_2 then F ← F ∪ {q_new}
11             else Q_aux ← Q_aux ∪ {q_new}
12         foreach transition (q_2^s, e_2, q_2^d) ∈ δ_2 do
13             q_new ← (q_1^s, q_2^d); δ ← δ ∪ {((q_1^s, q_2^s), e_2, q_new)}; Q ← Q ∪ {q_new}
14             if e_2 is a sending event ∨ q_new ∈ F_1 x F_2 then F ← F ∪ {q_new}
15             else Q_aux ← Q_aux ∪ {q_new}
16     A_1 ← (Q, Σ, δ, (q_01, q_02), F)
17 return A_1
```

Procedure `SplitFSA(`$\mathscr{A}$`)`

```
   /* Assume 𝒜 = (Q,Σ,δ,q0,F)                                          */
1  𝔸 ← ∅
2  foreach qf ∈ F do
       /* Get set Qrcl of all states from which qf can be reached, incl. qf  */
3      Qrcl ← ReverseClosure(qf)
4      δ' ← {(q1,e,q2) ∈ δ : q1,q2 ∈ Qrcl}
5      Σ' ← {e : (q1,e,q2) ∈ δ'}
6      𝔸 ← 𝔸 ∪ {(Qrcl,Σ',δ',q0,{qf})}
7  return 𝔸
```

Procedure `GetPosetsForObservedTT`$(p, v_0, E_{\text{ext}}, E_{\text{tt}}, \prec_{\text{tt}})$

1 $\Upsilon \leftarrow \emptyset$
2 **if** v_0 *is an activity node* **then**
  `/* Get the posets of` v_0 `that have the most common events with the triggering trace. Only one iteration of loops is considered. */`
3 $PS_{\text{match}} \leftarrow \{(E_1, \prec_1) \in [\![SD_{v_0}]\!]_{\text{SD}} : \nexists (E_2, \prec_2) \in ([\![SD_{v_0}]\!]_{\text{SD}} - \{(E_1, \prec_1)\}), \left|E_{\text{tt}} \cap E_2^p\right| > \left|E_{\text{tt}} \cap E_1^p\right|\}$
4 $PS_{\text{tail}} \leftarrow \emptyset$
5 $E'_{\text{ext}} \leftarrow E_{\text{ext}} - \{R_0^p : (R_0 \cup S_0, \prec_0) \in PS_{\text{match}}\}$
6 **if** $E'_{ext} \neq \emptyset$ **then**
7 **foreach** $v \in V$ **do**
8 **if** v *is an activity node* **then** $visited[ps_v] \leftarrow false$, for each poset ps_v of v
9 **else** $visited[v] \leftarrow false$
10 $PS_{\text{tail}} \leftarrow VisitSuccessorRP(p, u, E'_{\text{ext}}, (\emptyset, \emptyset))$ `// u is` v_0`'s successor`
11 **foreach** $(R_0 \cup S_0, \prec_0) \in PS_{match}$ **do**
12 $Min \leftarrow \{s \in min((S_0, \prec_0)) : loc(s) = p\}$ `// Minimum sendings of p in` ps_0
13 $F \leftarrow \{r \in E_{\text{ext}} \cap R_0^p : \nexists s \in Min, s \prec_0 r\}$
14 $\prec_{\text{ott}} \leftarrow \prec_{\text{tt}} \cup \{(r_1, r_2) \in \prec_0 : (r_1, r_2 \in F) \vee (r_1 \in E_{\text{tt}} \cap R_0^p \wedge r_2 \in F)\}$
15 $\prec_{\text{ott}} \leftarrow \prec_{\text{ott}} \cup \{(r_1, r_2) : r_1 \in (E_{\text{tt}} - R_0^p) \wedge r_2 \in F \wedge EventsInRace[r_1][r_2] = \emptyset\}$
16 $synched \leftarrow CheckSynchronization(p, v_0, (R_0 \cup S_0, \prec_0))$
17 $\Upsilon \leftarrow \Upsilon \cup UpdatePoset(synched, PS_{\text{tail}}, \prec_{\text{ott}}, E_{\text{tt}}, F, E'_{\text{ext}}, R_0^p, Min)$
18 **else**
  `/*` v_0 `is a fork node */`
19 $v_{\text{join}} \leftarrow null$ `// Global variable updated inside` *VisitSuccessorWC*
20 $(SynchPaths, UnsynchPaths) \leftarrow MapForkWC(v_0, \{p\}, (\emptyset, \emptyset))$
  `/* Get the posets in` $PS_{\texttt{fork}}$ `that have the most common events with the triggering trace */`
21 $PS_{\text{fork}} \leftarrow SynchPaths \cup UnsynchPaths$
22 $PS_{\text{match}} \leftarrow \{(seq, (E_1, \prec_1), \mathscr{R}) \in PS_{\text{fork}} : \nexists (E_2, \prec_2) \in (PS_{\text{fork}} - \{(seq, (E_1, \prec_1), \mathscr{R})\}), \left|E_{\text{tt}} \cap E_2^p\right| > \left|E_{\text{tt}} \cap E_1^p\right|\}$
23 $PS_{\text{tail}} \leftarrow \emptyset$
24 $E'_{\text{ext}} \leftarrow E_{\text{ext}} - \{R_0^p : (seq, (R_0 \cup S_0, \prec_0), \mathscr{R}) \in PS_{\text{match}}\}$
25 **if** $E'_{ext} \neq \emptyset$ **then**
26 **foreach** $v \in V$ **do**
27 **if** v *is an activity node* **then** $visited[ps_v] \leftarrow false$, for each poset ps_v of v
28 **else** $visited[v] \leftarrow false$
29 $PS_{\text{tail}} \leftarrow VisitSuccessorRP(p, u, E'_{\text{ext}}, (\emptyset, \emptyset))$ `//` u `is` $v_{\texttt{join}}$`'s successor`
30 **foreach** $(seq, (R_0 \cup S_0, \prec_0), \mathscr{R}) \in PS_{\text{match}}$ **do**
31 $Min \leftarrow \{s \in min((S_0, \prec_0)) : loc(s) = p\}$ `// Minimum sendings of p in` ps_0
32 $F \leftarrow \{r \in E_{\text{ext}} \cap R_0^p : \nexists s \in Min, s \prec_0 r\}$
33 $\prec_{\text{ott}} \leftarrow \prec_{\text{tt}} \cup \{(r_1, r_2) \in \prec_0 : (r_1, r_2 \in F) \vee (r_1 \in E_{\text{tt}} \cap R_0^p \wedge r_2 \in F)\}$
34 $\prec_{\text{ott}} \leftarrow \prec_{\text{ott}} \cup \{(r_1, r_2) : r_1 \in (E_{\text{tt}} - R_0^p) \wedge r_2 \in F \wedge EventsInRace[r_1][r_2] = \emptyset\}$
35 $\Upsilon \leftarrow \Upsilon \cup UpdatePoset((\mathscr{R} = \emptyset), PS_{\text{tail}}, \prec_{\text{ott}}, E_{\text{tt}}, F, E'_{\text{ext}}, R_0^p, Min)$
36 **return** Υ

Procedure `UpdatePoset`$(synched, PS_{\text{tail}}, \prec_{\text{ott}}, E_{\text{tt}}, F, E_{\text{ext}}, R_0^p, Min)$

$\Upsilon \leftarrow \emptyset$
if $PS_{\text{tail}} = \emptyset$ **then**
  $\Upsilon \leftarrow \Upsilon \cup \{(E_{\text{tt}} \cup F, \prec_{\text{ott}})\}$
else
  if $!synched$ **then**
    foreach $((E_{\text{tail}}, \prec_{\text{tail}}), \text{synchB}) \in PS_{\text{tail}}$ **do**
      $\prec_{\text{ott}} \leftarrow \prec_{\text{ott}} \cup \prec_{\text{tail}} \cup \{(r_1, r_2) : r_1 \in E_{\text{tt}} \cup F \wedge r_2 \in E_{\text{tail}} \wedge EventsInRace[r_1][r_2] = \emptyset\}$
      $\Upsilon \leftarrow \Upsilon \cup \{(E_{\text{tt}} \cup F \cup E_{\text{tail}}, \prec_{\text{ott}})\}$
  else if $\exists r \in E_{\text{tt}} \cap R_0^p$ *such that*
  $\forall s \in Min, s \not\prec_0 r$ *and* $EventsInRace[r][r'] \neq \emptyset$, *for any* $r' \in E_{\text{ext}}$) **then**
    foreach $((E_{\text{tail}}, \prec_{\text{tail}}), \text{synched}) \in PS_{\text{tail}}$ **do**
      $E_{\text{tail}} \leftarrow \{r_2 \in E_{\text{tail}} : EventsInRace[r_1][r_2] \neq \emptyset$ for any $r_1 \in E_{tt} \cap R_0^p\}$
      $\prec_{\text{ott}} \leftarrow \prec_{\text{ott}} \cup (\prec_{\text{tail}} \cap (E_{\text{tail}} \times E_{\text{tail}}))\}$
      $\prec_{\text{ott}} \leftarrow \prec_{\text{ott}} \cup \{(r_1, r_2) : r_1 \in E_{tt} \cap R_0^p \wedge r_2 \in E_{\text{tail}} \wedge EventsInRace[r_1][r_2] = \emptyset\}$
      $\Upsilon \leftarrow \Upsilon \cup \{(E_{\text{tt}} \cup F \cup E_{\text{tail}}, \prec_{\text{ott}})\}$
  else
    $\Upsilon \leftarrow \Upsilon \cup \{(E_{\text{tt}} \cup F, \prec_{\text{ott}})\}$
return Υ

Procedure `VisitSuccessorRP`$(p, v, E_{\text{ext}}, ps_{\text{aux}})$

```
1  if v is an activity node then
2      if p participates in v then
3          ps_old ← ps_aux; ϒ ← ∅
4          foreach poset ps_v = (R_v ∪ S_v, ≺_v) of v such that !visited[ps_v] do
5              ps_aux ← ps_old; visited[ps_v] ← true
               /* Consider ps_aux = (E_aux, ≺_aux) and ps_v = (R_v ∪ S_v, ≺_v)   */
6              F ← {r ∈ R_v^p ∩ E_ext}
7              E_ext ← E_ext − F
8              E_aux ← E_aux ∪ F
9              ≺_aux ← ≺_aux ∪ {(r1, r2) : r1 ∈ E_aux, r2 ∈ F, EventsInRace[r1][r2] = ∅}
10             ≺_aux ← ≺_aux ∪ {(r1, r2) ∈ ≺_v : r1, r2 ∈ F}
11             if E_ext = ∅ then
12                 ϒ ← ϒ ∪ {(ps_aux, false)}
13             else
14                 if CheckSynchronization(p, v, ps_v) = true then
15                     ϒ ← ϒ ∪ {(ps_aux, true)}
16                 else
17                     ϒ ← ϒ ∪ VisitSuccessorRP(p, u, E_ext, ps_aux) // u is v's successor
18             visited[ps_v] ← false
19     else
20         ϒ ← VisitSuccessorRP(p, u, E_ext, ps_aux) // u is v's successor
21     return ϒ // Backtrack
22 else
       /* Control node                                                          */
23     if v is a final node then
24         return {(ps_aux, false)}
25     else if v is a join node then
26         v_join ← v // Global variable used inside MapForkRP()
27         return {(ps_aux, false)}
28     else if v is a fork node and !visited[v] then
29         visited[v] ← true
30         ϒ ← TraverseForkRP(p, v, E_ext, ps_aux)
31         visited[v] ← false
32         return ϒ
33     else
34         ps_old ← ps_aux
35         foreach u successor of v do
36             if E_ext ≠ ∅ then
37                 ps_aux ← ps_old
38                 ϒ_aux ← VisitSuccessorRP(p, u, E_ext, ps_aux)
39                 E_ext ← E_ext − {E_aux : (E_aux, ≺_aux) ∈ ϒ_aux}
40                 ϒ ← ϒ ∪ ϒ_aux
41         return ϒ // Backtrack
```

Procedure `TraverseForkRP`$(p, v_{\text{fork}}, E_{\text{ext}}, ps_{\text{aux}})$

```
1  v_join ← null // Global variable updated when a join node is visited
2  E_ext^new ← E_ext
3  foreach successor u of v_fork do
4      if E_ext^new ≠ ∅ then
5          ϒ ← VisitSuccessorRP(p, u, E_ext, (∅, ∅))
6          ϒ_fork ← ϒ_fork ∪ ϒ
7          E_ext^new ← E_ext^new − {E : ((E, ≺), synchB) ∈ ϒ, for any ≺, synchB}
8  ϒ ← ∅
9  foreach ϒ_path ∈ getCombs(ϒ_fork) do
10     (E_forkP, ≺_forkP) ← CausalOrderPar({ps : (ps, synchB) ∈ ϒ_path, for any synchB})
11     ≺_aux ← ≺_aux ∪ ≺_forkP ∪{(r1, r2) : r1 ∈ E_aux ∧ r2 ∈ E_forkP ∧ EventsInRace[r1][r2] = ∅}
12     E_aux ← E_aux ∪ E_forkP
13     if (v_join = null ) ∨ (E_ext^new = ∅) ∨ (∃(ps, true) ∈ ϒ_path, for any ps) then
14         ϒ ← ϒ ∪ {((E_aux, ≺_aux), true)}
15     else
16         v_join_succ ← successor of v_join
17         ϒ ← ϒ ∪ VisitSuccessorRP(p, v_join_succ, E_ext − E_forkP, (E_aux, ≺_aux))
18 return ϒ
```

Procedure `CheckSynchronization`(p, v, ps_v)

```
1  synched ← false
2  if p has a sending in ps_v then
3      synched ← true
4      foreach successor activity u of v do
5          foreach poset ps_u of u do
               /* We assume ps_v = (S_v ∪ R_v, ≺_v)                          */
6              I ← {loc(e) : e ∈ min(ps_u)} // Initiating roles of ps_u
               /* Get max events in ps_v of I roles                          */
7              foreach q ∈ I do Max[q] ← {e : e ∈ max(ps_v) ∧ loc(e) = q}
8              Min ← {s : s ∈ min((S_v, ≺_v)) ∧ loc(s) = p}// Min sendings in ps_v of p
9              if ∃q ∈ I such that ∀m ∈ Max[q], ∄s ∈ Min, s ≺_v m then
10                 synched ← false
11             else
12                 visited[ps_u] ← true
13 return synched
```

6 Conclusions

We have outlined a collaboration-oriented service specification approach, where UML 2 collaborations are used to specify services. The behavior of elementary collaborations is described by means of UML sequence diagrams, while the behavior of composite collaborations is described with help of a choreography graph (following the notation of UML activity diagrams) that defines the execution order of its sub-collaborations. We have provided a formal syntax and semantics to choreography graphs and sequence diagrams in terms of partial orders.

We have discussed realizability of choreographies in terms of the composition operators: weak and strong sequence, alternative, interruption and parallel. For each composition operator we have studied the problems that can lead to difficulties of realization. We have investigated the actual nature of these problems and discussed possible solutions to prevent or remedy them. The result of our study is a better understanding of the actual nature of realizability problems. Not surprisingly, we have seen that implicit concurrency and competing initiatives are at the heart of most problems. The send-causality property identified in this paper helps to build specifications that are more intuitive and less prone to conflicts, since it forces concurrency to be explicitly specified (i.e. by means of parallel composition or interruption). We have shown that some problems can already be detected at an abstract collaboration level, without needing to look into detailed interactions. We have also shown that generic solutions to the discussed problems are not valid. The same type of problem may require different resolutions in different contexts.

Finally, we have presented a set of algorithms for the detection of the problems discussed in this paper, and are currently working on their implementation.

References

[AEY00] Rajeev Alur, Kousha Etessami, and Mihalis Yannakakis. Inference of message sequence charts. In *22nd Int. Conf. on Software Engineering (ICSE'00)*, 2000.

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

[AHP96] Rajeev Alur, Gerard J. Holzmann, and Doron Peled. An analyzer for message sequence charts. *Software - Concepts and Tools*, 17(2):70–77, 1996.

[AHU74] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The design and analysis of computer algorithms*. Addison-Wesley, 1974.

[BAL97] Hanene Ben-Abdallah and Stefan Leue. Syntactic detection of process divergence and non-local choice in message sequence charts. In *Proc. 2nd Int. Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'97)*, 1997.

[BG86] Gregor Bochmann and Reinhard Gotzhein. Deriving protocol specifications from service specifications. In *Proc. of ACM SIGCOMM Symposium*, pages 148–156, 1986.

[BM03] Nicolas Baudru and Rémi Morin. Safe implementability of regular message sequence chart specifications. In *Proc. of ACIS 4th Intl. Conf. on Soft. Eng., Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD'03)*, pages 210–217, 2003.

[BM05] Rolv Bræk and Geir Melby. *Model Driven Service Engineering*, chapter of Model-driven Software Development. Volume II of Research and Practice in Software Engineering. Springer, 2005.

[Boc78] Gregor Bochmann. Finite state description of communication protocols. *Computer Networks*, 2:361–372, 1978.

[Bræ79] Rolv Bræk. Unified system modeling and implementation. In *International Switching Symposium (ISS)*. ISS Committee, 1979.

[BS05] Yves Bontemps and Pierre-Yves Schobbens. The complexity of live sequence charts. In *Proc. of 8th Intl. Conf. on Foundations of Software Science and Computational Structures (FoSSaCS '05)*, pages 364–378, 2005.

[CB06a] Humberto N. Castejón and Rolv Bræk. A collaboration-based approach to service specification and detection of implied scenarios. In *Proc. of 5th int. workshop on Scenarios and state machines: models, algorithms and tools (SCESM'06)*. ACM Press, 2006.

[CB06b] Humberto N. Castejón and Rolv Bræk. Formalizing collaboration goal sequences for service choreography. In *Proc. of the 26th IFIP WG 6.1 Intl. Conf. on Formal Methods for Networked and Distributed Systems (FORTE'06)*, volume 4229 of *LNCS*, pages 275–291, Paris, France, September 2006. Springer-Verlag.

[CKS05] Chien-An Chen, Sara Kalvala, and Jane Sinclair. Race conditions in message sequence charts. In *Proc. of 3rd Asian Symposium on Programming Languages and Systems (APLAS'05)*, volume 3780 of *LNCS*, pages 195–211. Springer, 2005.

[Erl05] Thomas Erl. *Service Oriented Architecture: Concepts, Technology and Design*. Number ISBN 0-13-185858-0. Prentice Hall, 2005.

[FK01] Kathi Fisler and Shriram Krishnamurthi. Modular verification of collaboration-based software designs. In *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 152–163. ACM Press, 2001.

[GMSZ06] Blaise Genest, Anca Muscholl, Helmut Seidl, and Marc Zeitoun. Infinite-state high-level MSCs: Model-checking and realizability. *J. Comput. Syst. Sci.*, 72(4):617–647, 2006.

[GY84] Mohamed G. Gouda and Yao-Tin Yu. Synthesis of communicating finite state machines with guaranteed progress. *IEEE Trans. on Communications*, Com-32(7):779–788, July 1984.

[Hél01] Loïc Hélouët. Some pathological message sequence charts, and how to detect them. In *10th Intl. SDL Forum*, volume 2078 of *LNCS*, pages 348–364. Springer-Verlag, 2001.

[HJ00] Loïc Hélouët and Claude Jard. Conditions for synthesis of communicating automata from HMSCs. In *Proc. of 5th Intl. Workshop on Formal Methods for Industrial Critical Systems (FMICS'00)*. GMD FOKUS, 2000.

[HMU00] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison Wesley, 2000.

[IT99] ITU-T. *ITU Recommendation Z.Z.120: "Message Sequence Chart (MSC-2000)".* ITU, Geneva, 1999.

[IT00] ITU-T. *ITU Recommendation Z.100: "The Specification and Description Language (SDL)".* ITU, Geneva, 2000.

[Kie97] Astrid Kiehn. Observing partial order runs of petri nets. In *Foundations of Computer Science: Potential - Theory - Cognition, to Wilfried Brauer on the occasion of his sixtieth birthday*, pages 233–238, London, UK, 1997. Springer-Verlag.

[KL98] J. P. Katoen and L. Lambert. Pomsets for message sequence charts. In H. König and P. Langendörfer, editors, *Formale Beschreibungstechniken fuer verteilte Systeme, 8. GI/ITG-Fachgespraech, Cottbus, Germany*, pages 197–207. Shaker Verlag, 1998.

[KM03] Ingolf H. Krüger and Reena Mathew. Component synthesis from service specifications. In *2003 Dagstuhl Workshop on Scenarios: Models, Transformations and Tools*, volume 3466 of *LNCS*, pages 255–277. Springer, 2005.

[KZ05] Ferhat Khendek and Xiao Jun Zhang. From MSC to SDL: Overview and an application to the autonomous shuttle transport system. In *2003 Dagstuhl Workshop on Scenarios: Models, Transformations and Tools*, volume 3466 of *LNCS*, pages 228–254. Springer, 2005.

[MGR05] Arjan J. Mooij, Nicolae Goga, and Judi Romijn. Non-local choice and beyond: Intricacies of MSC choice nodes. In *Proc. Intl. Conf. on Fundamental Approaches to Soft. Eng. (FASE'05)*, volume 3442 of *LNCS*. Springer, 2005.

[Mit05] Bill Mitchell. Resolving race conditions in asynchronous partial order scenarios. *IEEE Trans. Softw. Eng.*, 31(9):767–784, 2005.

[MP00] Anca Muscholl and Doron Peled. Analyzing message sequence charts. In *SAM*, pages 3–17, 2000.

[MRW06] Arjan Mooij, Judi Romijn, and Wieger Wesselink. Realizability criteria for compositional msc. In *Proc. of 11th Intl. Conf. on Algebraic Methodology and Software Technology (AMAST'06)*, volume 4019 of *LNCS*. Springer, 2006.

[Mus00] Anca Muscholl. Compositional issues on message sequence charts. In *Proc. Workshop on Logic and Algebra in Concurrency*, 2000.

[OMG07] Object Management Group (OMG). *UML 2.1.1 Superstructure Spec.*, February 2007.

[ON05] Akimitsu Ono and Shin-Ichi Nakano. Constant time generation of linear extensions. In *15th Intl. Symp. on Fundamentals of Computation Theory (FCT'05)*, pages 445–453, 2005.

[RAB+92] T. Reenskaug, E.P. Andersen, A.J. Berre, A. Hurlen, A. Landmark, O.A. Lehne, E. Nordhagen, E. Ness-Ulseth, G. Oftedal, A.L. Skaar, and P. Stenslet. OORASS: Seamless support for the creation and maintenance of object-oriented systems. *Journal of Object-oriented Programming*, 5(6):27–41, 1992.

[RGG01] Frank Rößler, Birgit Geppert, and Reinhard Gotzhein. Collaboration-based design of SDL systems. In *Proc. of the 10th Intl. SDL Forum*, volume 2078 of *LNCS*, pages 72–89. Springer-Verlag, 2001.

[RWL96] Trygve Reenskaug, P. Wold, and O. A. Lehne. *Working with Objects: The OOram Software Engineering Method*. Prentice Hall, 1996.

[San00] Richard Sanders. Implementing from SDL. *Telektronikk*, 96(4), 2000.

[SCKB05] Richard Torbjørn Sanders, Humberto N. Castejón, Frank Alexander Kraemer, and Rolv Bræk. Using UML 2.0 collaborations for compositional service specification. In *ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, volume 3713 of *LNCS*, pages 460–475, Montego Bay, Jamaica, October 2005. Springer-Verlag.

[UKM04] Sebastián Uchitel, Jeff Kramer, and Jeff Magee. Incremental elaboration of scenario-based specifications and behavior models using implied scenarios. *ACM Trans. Softw. Eng. Methodol.*, 13(1):37–85, 2004.

[Woo87] Derick Wood. *Theory of Computation*. John Wiley & Sons, Inc., New York, 1987.

A Propositions and Proofs

We show here that in sequence diagrams with both the send-causality and the non-crossing messages properties, race conditions may only occur between two or more consecutive receiving events. We introduce first some useful propositions, which will help on the demonstration.

Proposition A.1. *In a basic sequence diagram with the send-causality property all sending events are causally ordered (i.e. there is a total causal order on sending events).*

Proof. We prove it for out-of-order delivery semantics, so the result is also valid for in-order delivery semantics.

Consider two consecutive sending events according to $<_m$, that is, $s, s' \in S, s <_m s' \wedge \nexists s'' \in S, s <_m s'' <_m s'$. Then, from the definition of send-causal sequence diagram (Definition 4.5), we have that either $loc(s') = loc(s)$ or $loc(s') = loc(rcv(s))$. If $loc(s') = loc(s) = p$, then $s <_p s'$, and from the definition of causal order with out-of-order delivery (Definition 3.2), we conclude $s \prec_{nf} s'$. Otherwise, if $loc(s') = loc(rcv(s)) = p$, then $rcv(s) <_p s'$. Again, by Definition 3.2, we conclude $s \prec_{nf} s'$. Since any two consecutive sending events are causally ordered, and by transitivity of $\prec_{nf}$, we conclude that all sending events are causally ordered. □

Corollary A.2. *A send-causal basic sequence diagram has a unique initiating event.*

Note that, in the absence of parallel composition, send-causality imposes a total causal order of sending events for each alternative behavior. If a sequence diagram describes a parallel composition by means of a *par* construct, the sending events within each operand of the construct are related by a total causal order. In addition, the sending events preceding (resp. succeeding) the *par* construct are causal predecessors (resp. successors) of all sending events within the *par* construct.

Now we demonstrate that in a send-causal sequence diagram, if a receiving event $r \in R$ is specified to happen after a sending event $s \in S$ on the same lifeline $p \in \mathscr{P}$ (i.e. $s <_p r$), then r is always causally dependent on s, with independence of the communication architecture (i.e. r will always happen after s in a any realized system).

Proposition A.3. *In a send-causal sequence diagram satisfying the non-crossing messages property, the following is always true: given a sending event $s \in S$ and a receiving event $r \in R$ located on the same lifeline $p \in \mathscr{P}$ (i.e. $loc(s) = loc(r) = p$), we have that $s <_p r \Rightarrow s \prec_{nf} r \wedge s \prec_f r$.*

Proof. We prove it for the out-of-order delivery causal order ($\prec_{nf}$), since it corresponds to a communication architecture without restrictions. The results will then be valid for any more restrictive order (e.g. $\prec_f$). We consider first the case where s and r are located within the same basic sequence sub-diagram. By Proposition A.1, we know that $s \prec_{nf} snd(r)$. We conclude, therefore, that $s \prec_{nf} r$.

We consider now two basic sequence sub-diagrams, SD_1 and SD_2, such that $SD_1 \, seq \, SD_2$. We assume s is located in SD_1 and r is located in SD_2. We know that:

(i) Either s is a maximum sending event of SD_1 (i.e. $s \in \mathscr{T}_s, \mathscr{T}_s \in term_{snd}(SD_1)$), or, by Proposition A.1, we have that $s \prec_{nf} s_t, \forall s_t \in \mathscr{T}_s, \mathscr{T}_s \in term_{snd}(SD_1)$.

(ii) Either $snd(r)$ is a minimum event of SD_2 (i.e. $snd(r) \in \mathscr{I}, \mathscr{I} \in init(SD_2)$), or, by Proposition A.1, we have that $s_i \prec_{nf} snd(r), \forall s_i \in \mathscr{I}, \mathscr{I} \in init(SD_2)$.

By Definition 4.5, we also know that $\forall \mathscr{T}_s \in term_{snd}(SD_1), \forall \mathscr{I} \in init(SD_2), \forall s_t \in \mathscr{T}_s, \forall s_i \in \mathscr{I}, loc(s_i) = loc(s_t) \vee loc(s_i) = loc(rcv(s_t))$. With this information, and applying the same reasoning as the one used in the proof of Proposition A.1, we conclude that $s \prec_{nf} r$.

The above results can be easily generalized, by induction on the composite structure of the sequence diagram, to prove that in all cases $s \prec_{nf} r$. □

Given Proposition A.3 and the fact that races may exist between consecutive receiving events (see, e.g., the race between e_4 and e_6 in Fig. 5(a)) we have the result we were looking for:

Proposition A.4. *In a sequence diagram satisfying the send-causality property and the non-crossing messages property, a potential race condition exists between two receiving events r_1 and r_2, located at the same lifeline p, if $r_1 <_p r_2$ and there is not a sending event $s \in S$ such that $r_1 <_p s <_p r_2$.*

A.1 Race Conditions in Send-Causal Sequence Diagrams

We study now the necessary conditions for a race to actually exist in a send-causal sequence diagram. Such conditions depend on the communication architecture. We consider two architectures, one with out-of-order delivery channels and other with in-order delivery channels.

Race conditions with in-order delivery channels

Messages cannot overtake each other in channels with in-order delivery. Therefore, there are not races between receiving events located on the same lifeline p if their associated sending events are both located on the same lifeline q. However, a race exists if the sending events are located on different lifelines and there is no sending event located on p between the receiving events.

Proposition A.5. *Given a sequence diagram satisfying the send-causality property and the non-crossing messages property, and a communication architecture with in-order delivery channels, a race condition exists between two receiving events, r and r', located on the same lifeline $p \in \mathscr{P}$, iff $r <_p r' \wedge loc(snd(r)) \neq loc(snd(r')) \wedge \nexists s \in S, r <_p s <_p r'$.*

Proof. ($\Longleftarrow$) We assume that $r <_p r' \wedge loc(snd(r)) \neq loc(snd(r')) \wedge \nexists s \in S, r <_p s <_p r'$ and prove that there is a race between r and r' (i.e. $r \not\prec_f r'$). We do this by contradiction. Let us assume that $r \prec_f r'$. Then we should have that $r \prec_f snd(r')$. We

recall that, according to Definition 3.3, $\prec_f = (<_f)^*$. It is easy to see that $r \not<_f snd(r')$, since r and $snd(r')$ do not satisfy any of the conditions on Definition 3.3. Therefore, there must exist one or more events such that $r <_f e_1 <_f \ldots e_n <_f snd(r')$. Since $loc(snd(r)) \neq loc(snd(r'))$, the relation $r <_f e_1$ can only be true if $e_1 \in S \wedge r <_p e_1$. This contradicts our initial assumption: $\nexists s \in S, r <_p s$; so we conclude that $r \not<_f r'$.

($\Longrightarrow$) We consider two receiving events, r and r', that are located on the same lifeline $p \in \mathscr{P}$ and that are in race, so that $r \not<_f r'$. By the definition of race condition, we know that $r <_p r'$. We also know that in-order delivery semantics do not allow message overtaking, so if $loc(snd(r)) = loc(snd(r'))$ there cannot be a race. Therefore it must be $loc(snd(r)) \neq loc(snd(r'))$. We just need to prove that $\nexists s \in S, r <_p s <_p r'$. Let us assume that $\exists s \in S, r <_p s <_p r'$. Then, by Definition 3.3, we have that $r \prec_f s$, and by Proposition A.3, we have that $s \prec_f r'$. This implies $r \prec_f r'$, which is a contradiction. Therefore $\nexists s \in S, r <_p s <_p r'$. □

Race conditions with out-of-order delivery channels.

With out-of-order delivery channels races always exist between receiving events located on the same lifeline p if there is no sending event located also p between the receiving events.

Proposition A.6. *Given a sequence diagram satisfying the send-causality property and the non-crossing messages property, and a communication architecture with out-of-order delivery channels, a race condition exists between two receiving events, r and r', located on the same lifeline $p \in \mathscr{P}$, iff $r <_p r' \wedge \nexists s \in S, r <_p s <_p r'$.*

Proof. Since the causal order imposed by out-of-order delivery channels is less restrictive than the causal order imposed by in-order delivery channels (i.e. $\prec_{nf} \subseteq \prec_f$), races will exists with out-of-order delivery channels whenever they exist with in-order delivery channels. The case of races between between receiving events associated to messages sent by different lifelines is therefore proved by A.5. We just need to prove that there are also races between receiving events associated to messages sent by the same lifeline. This can be easily done following the same reasoning as with Proposition A.5, and having into account that messages may overtake each other in an out-of-order delivery channel. □

B Automata Theory

A finite state automaton is a tuple $\mathscr{A} = (Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states, Σ is a set of transition labels, $\delta \subseteq Q \times \Sigma \times Q$ is a set of transitions, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is the set of final (or accepting) states.

B.1 Converting an FSA into a regular expression

There exist several methods for the conversion of an FSA into an equivalent regular expression (i.e. a regular expression accepting the same language as the FSA). Here

we describe the state-elimination technique from [Woo87].

The intuitive idea behind the state-elimination technique is to bypass an state $q \in Q - \{q_0, q_f\}$ by replacing that state, together with its incoming, outgoing and self-looping transitions, with new transitions. These new transitions are labeled with regular expressions, such that the resulting automaton accepts the same language as the original one.

As input for the state-elimination process we assume an automaton $\mathscr{A} = (Q, \Sigma, \delta, q_0, q_f)$ with the following properties (which facilitate the elimination process):

- $\mathscr{A}$ has one single initial state, and this state has no input or self-looping transitions. If the original automaton does not fulfill this requisite, a new initial state q_0' can be added and connected to q_0 by an ε-transition (i.e. $Q' = Q \cup \{q_0'\}$, $\delta' = \delta \cup \{(q_0', \varepsilon, q_0)\}$).
- $\mathscr{A}$ has one single final state, and this state has no output or self-looping transitions. If this is not the case for the original automaton, the original final states are converted into normal states and connected to a new final state q_f' by ε-transitions (i.e. $Q' = Q \cup \{q_f'\}$, $\delta' = \delta \cup \{(q, \varepsilon, q_f') : q \in F\}$, $F' = \{q_f'\}$).
- Each state $q \in Q - \{q_0, q_f\}$ has a unique self-looping transition $(q, \beta, q) \in \delta$. This is a merely accessory property and required to ease the explanation below. Note that if a state $q \in Q - \{q_0, q_f\}$ has not self-looping transitions, we can assume a self-looping silent transition (i.e. $\beta = \varepsilon$). And if q has several self-looping transitions, we can merge them.

Formally, a step of the state elimination technique can be seen as the transformation of an automaton $\mathscr{A} = (Q, \Sigma, \delta, q_0, q_f)$ into a new automaton $\mathscr{A}' = (Q - \{q\}, \Sigma', \delta', q_0, q_f)$, where $q \in Q - \{q_0, q_f\}$ is the state being eliminated. $\mathscr{A}'$ is then used as input for the next step. The elimination process is repeated until we obtain an automaton $\mathscr{A}_E = (\{q_0, q_f\}, \Sigma_E, \{(q_0, E, q_f)\}, q_0, q_f)$ that consists of exactly two states, the initial and final ones, and one transition between them. Such transition is labeled with a regular expression E, which accepts exactly the same language as the original automaton $\mathscr{A}$.

For the elimination of a state $q \in Q - \{q_0, q_f\}$ we proceed as follows. For each input transition $(p_i, u_i, q) \in \delta$, with $1 \leq i \leq n$ and $n \geq 1$, each output transition $(q, v_j, r_j) \in \delta$, with $1 \leq j \leq m$ and $m \geq 1$, and the self-looping transition $(q, w, q) \in \delta$, we create a new transition $(p_i, u_i \cdot w^* \cdot v_j, r_j)$. We then obtain Σ' and δ' as indicated below:

$$\begin{aligned}
\delta' &= \delta - (\{(p_i, u_i, q), (q, v_j, r_j) : 1 \leq i \leq n \wedge n \geq 1 \wedge 1 \leq j \leq m \wedge m \geq 1\} \cup \{(q, w, q)\}) \\
&\quad \cup \{(p_i, u_i \cdot w^* \cdot v_j, r_j) : 1 \leq i \leq n \wedge n \geq 1 \wedge 1 \leq j \leq m \wedge m \geq 1\} \\
\Sigma' &= \Sigma \cup \{u_i \cdot w^* \cdot v_j : 1 \leq i \leq n \wedge n \geq 1 \wedge 1 \leq j \leq m \wedge m \geq 1\}
\end{aligned}$$

If δ' contains more than one transition between any two states, we merge those transitions. Formally, given two transitions $(q_1, u, q_2), (q_1, v, q_2) \in \delta'$, we remove them from δ' and add a new transition $(q_1, u \mid v, q_2)$. We also add the label $u \mid v$ to Σ'.

Figure 19(a) shows the automaton obtained after running the *ChoreographyTo-FSA* procedure on the choreography graph of Fig. 17(a). Figures 19(b) and 19(c) illustrate some steps of the state elimination technique applied to that automaton. The automaton in Fig. 19(b) results after eliminating states m_1, q_0, q_3, q_6 and q_8. Note that since the two first properties discussed above where not satisfied by the automaton in Fig. 19(a), a new initial state, q'_0, and a new final state, q_f, were added. After eliminating all states, except the initial and final ones, the automaton in Fig. 19(c) is obtained. Merging the transitions of that automaton would give a new automaton with a single transition, whose label would correspond to a regular expression accepting the same language as the FSA in Fig. 19(a).

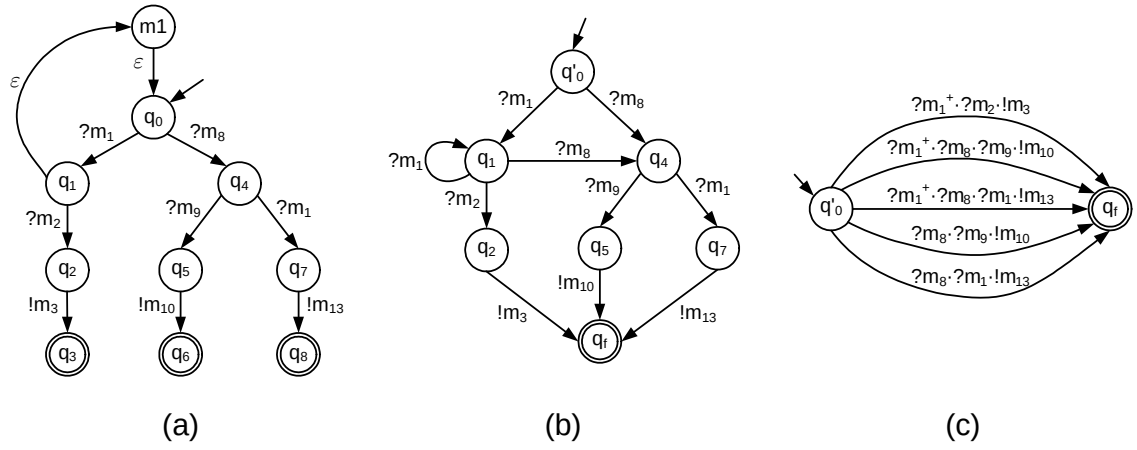

Figure 19: (a) FSA for the choreography of Fig. 17(a); (b) FSA resulting after eliminating all ε-transitions in (a); (c) FSA resulting after eliminating all non-initial and non-final states in (b)

B.2 Eliminating ε-transitions

To eliminate these ε-transitions the technique described in [HMU00] can be used. This technique consists of three basic steps:

a) Identify all states Q_ε with output ε-transitions.

b) For each state $p \in Q_\varepsilon$, find all states Q' that can be reached using only ε-transitions (this can be done with a DFS technique). For each state $q \in Q'$, if there is a transition to state r on input e (i.e. a non ε-transition), then create a new transition (p, e, r) from p to r on input e. If any state $q \in Q'$ is a final state, then make p a final state.

c) Remove all ε-transitions and all unreachable states.

Notes

[1]The second condition for the *non-crossing messages* property was wrongly formulated in the original technical report. In the original text it could be read

$$\forall e = \langle !m, p, p \rangle \, , \nexists e' \neq rcv(\langle !m, p, p \rangle), e <_p e'$$

We note that such condition would be right if $<_p$ was an immediate precedence relation, rather than a total order.