Direct Realizability

Humberto N. Castejón, Gregor von Bochmann, Rolv Bræk

1 Introduction

We consider here a UML 2 collaboration describing the structure of a service as a set of roles and the sub-collaborations that take place between them. Such sub-collaborations are represented as UML collaboration uses, and can themselves be composite collaborations or elementary collaborations, which are no further decomposed.

We assume the behavior of an elementary collaboration is described with the help of a UML 2 sequence diagram. The behavior of a composite collaboration is described as a choreography of its sub-collaborations. Such choreography is specified using an UML 2 activity diagram, where each nodes is a *CallBehaviorAction* that invokes the behavior associated with a collaboration type and made available via a collaboration use (in the diagram describing the structure of the composite collaboration).

Given a choreography, we are interested in finding a set of communicating role behaviors whose joint behavior is precisely the specified by the choreography. Whenever it is possible to find such set of role behaviors we say that the choreography is **direct realizability**. In Section 5 we define the notion of direct realizability of a choreography formally. To do that we first introduce, in Section 2, the formal syntax of sequence diagrams, and in Section 3 we present the formal syntax of choreography graphs and define the set of all traces (i.e. sequences of events) defined by a choreography. In Section 4 we introduce the notion of directly realized system, and define the set of all traces generated by the execution of such system. Finally, in Section 6 we present some propositions and their proofs.

2 Formal syntax of sequence diagrams

A basic sequence diagram defines a labeled directed acyclic graph that can be described by a tuple $bSD = (R,E,M,\Sigma,\lambda,\phi,rcv,<)$ where

- R is a set of lifelines (or roles)
- $E = E_! \cup E_?$ is a set of sending ($E_!$) and receiving ($E_?$) events
- M is a set of messages and Σ is a set of communication actions of the form $p!q(m)$ (read "p sends message m to q") and $p?q(m)$ (read "p receives message m from q"), with $p,q \in R$ and $m \in M$
- $\lambda : E \to \Sigma$ is a mapping that associates events with communication actions
- $\phi : E \to R$ is a mapping that associates events with the roles that perform them. We let $E^p = \{e \in E : \phi(e) = p, p \in R\}$
- $rcv : E_! \to E_?$ is a bijection that pairs up the sending and receiving events associated with the transmission of a message
- $<= ((\bigcup_{p \in R} <_p) \cup \{(e, rcv(e)) : e \in E_!\})^*$ is a (strict) partial order on E, called the *visual order*.

  For each $p \in R$, $<_p$ is a total order over the events performed by p.

Basic sequence diagrams can be composed to obtain more complex behaviors. In UML 2 this is possible by means of interaction operators. Here we consider four operators:

- Weak sequential composition (`seq`) of two sequence diagrams, which consists in their lifeline-by-lifeline concatenation, such that for each lifeline, the events of the first

diagram precede the events of the second diagram. Events on different lifelines are interleaved.

- Alternative composition (`alt`) of two sequence diagrams, which 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 visual orders.
- Parallel composition (`par`) of two sequence diagrams, where the events from both diagrams are interleaved.
- Iterative composition (`loop`) of a sequence diagram, which 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:

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

A composite sequence diagram will, in general, describe one or more alternative visual orders. The set Λ of alternative visual orders described by a composite sequence diagram SD is defined as follows:

$\Lambda = \{(E_1, <_1)\}$, if $SD = bSD_1$

$\Lambda = \{(<, E_1 \cup E_2) : <= (<_1 \cup <_2 \cup \{(e_1,e_2) \in E_1 \times E_2 : \phi(e_1) = \phi(e_2)\})^*, (<_1, E_1) \in \Lambda_1, (<_2, E_2) \in \Lambda_2\}$,
if $SD = SD_1 \texttt{ seq } SD_2$

$\Lambda = \{(<_1 \cup <_2, E_1 \cup E_2) : (<_1, E_1) \in \Lambda_1, (<_2, E_2) \in \Lambda_2\}$, if $SD = SD_1 \texttt{ par } SD_2$

$\Lambda = \Lambda_1 \bigcup \Lambda_2$, if $SD = SD_1 \texttt{ alt } SD_2$

$\Lambda = \bigcup_{\min \le n \le \max} \Lambda_1^n$, if $SD = \texttt{loop}(\min,\max)\, SD_1$, with $1 < \min \le \max$, and Λ_1^n is the set of visual orders described by $SD_1^n = SD_1 \texttt{ seq } SD_1^{n-1}$, for $n > 1$, and $SD_1^n = SD_1$, for $n = 1$.

3 Formal syntax and traces of choreography graphs

We use UML 2 activity diagrams (although with modified semantics) to describe the choreography of the sub-collaborations of a composite collaboration. Formally, we define a choreography graph as a directed graph defined by the tuple $Ch = (V, I, \triangleright, \rightarrow \Omega, \mu)$, where

- V is a set of nodes that is partitioned into an initial node (v_0) and sub-sets of *CallBehaviorAction* nodes (V_{A}), control flow nodes (V_{C}), accept event actions (V_{E}) and final nodes (V_{F}). The set V_{C} is in turn partitioned into decision (V_{D}), merge (V_{M}), fork (V_{K}) and join (V_{J}) nodes.
- I is a set of interruptible regions (i.e. dashed regions containing nodes that can be interrupted).
- $\triangleright \subseteq I \times (I \cup V)$ is a hierarchy relation among interruptible regions and nodes.
- $\rightarrow \subseteq (\{v_0\} \cup V_{\mathrm{A}} \cup V_{\mathrm{C}} \cup V_{\mathrm{E}}) \times (V_{\mathrm{A}} \cup V_{\mathrm{C}} \cup V_{\mathrm{F}})$ is a set of directed edges between nodes, which is partitioned into normal ($\rightarrow_{\mathrm{n}}$) and interrupting edges ($\rightarrow_{\mathrm{i}}$). We write $u \rightarrow v$ if $(u,v) \in \rightarrow$. For convenience, given $u, v \in V_A$, we write $u \rightsquigarrow v$ if either $u \rightarrow v$ or $u \rightarrow w_1 \rightarrow \ldots \rightarrow w_n \rightarrow v$ and $w_i \notin V_{\mathrm{A}}, \forall i \in [1 \ldots n]$ (that is, it is possible to reach *v* from *u* without traversing any other CallBehaviorAction).

Interrupting edges must have its source node inside an interruptible region and its target node outside the region, that is, if $u \rightarrow_{\mathrm{i}} v$, then $\exists R \in I$ such that $R \triangleright u$ and not $R \triangleright v$.

- Ω is a set of labels of the form $cu.B$, where *cu* is the name of one of the collaboration-uses of *C*, and *B* is the sequence or activity diagram describing the behavior of the collaboration which *cu* represents an occurrence of.
- $\mu : V_{\mathrm{A}} \rightarrow \Omega$ is a mapping between CallBehaviorActions and labels.

A choreography graph, as defined above, allows specifying the following execution orderings:

- Sequential execution, described with help of activity edges, which have the meaning of weak sequencing.
- Parallel execution, described with help of fork and join nodes.
- Alternative execution, described with help of decision nodes.
- Interruption, described with help of interruptible regions and interrupting edges. The following constraint has to be obeyed in order to avoid deadlocks. If a CallBehaviorAction *X* is inside an interruptible region *R* and *X* leads to a join node *J* ("normal" path), then the interrupting edge leaving *R* must also eventually lead to *J* (and be merged with the "normal" path with help of a merge node, which should be directly connected to *J*). Obeying this constraint is not necessary if all other CallBehaviorActions that lead to *J* are also inside *R*. See Figure 1(a) for an example.

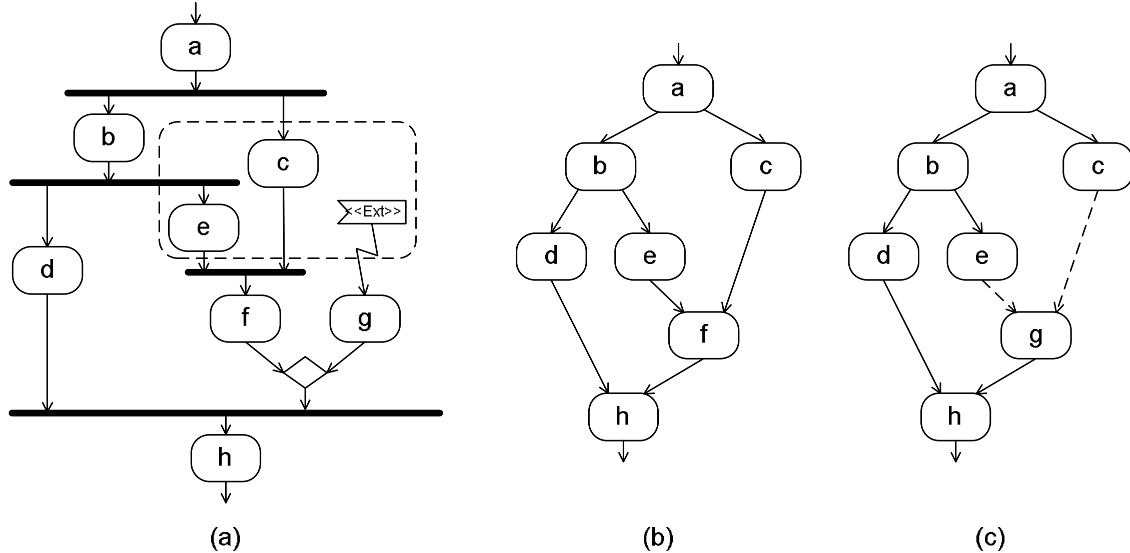

Figure 1. (a) A choreography graph and the two possible paths through it: (b) when the interruption does not happen; and (c) when it happens

Sequential, parallel and alternative execution orderings have the same semantics as the `seq`, `par` and `alt` composition operators defined for sequence diagrams in Section 2. To the semantics of interruption we need to introduce the notion of prefix of a visual order.

Definition (Prefix). A visual order $(E', <')$ is a prefix of a visual order $(E, <)$ iff

- $E' \subset E$
- if $e \in E'_{!}$, then $rcv(e) \in E'$; and if $e < f$ and $f \in E'$, then $e \in E'$, $\forall e, f$
- $<' = < \cap (E' \times E')$

We let $pre(E,<)$ denote the set of all prefixes of a visual order $(E,<)$, and for a set Λ of visual orders, we let $pre(\Lambda) = \{(E',<') \in pre(E,<) : (E,<) \in \Lambda\}$. For a sequence diagram *SD* describing a set Λ of visual orders, we let *prefix*(*SD*) be the sequence diagram describing the set $pre(\Lambda)$ of visual orders.

Given two CallBehaviorActions *n* and *m*, the interruption of *n* by *m* defines a sequence diagram $SD = prefix(SD_n)\ \texttt{seq}\ SD_m$, where SD_n (resp. SD_m) is the sequence diagram describing the behavior of the collaboration invoked by *n* (resp. *m*).

3.1 Traces

A choreography defines a set of traces, one for each possible *complete path* through the choreography, and for each possible ordering of events within that path. To simplify the definition of traces, we propose to flatten the choreography graph by eliminating the CallBehaviorActions that invoke a collaboration behavior defined by an activity diagram, and replacing them by the activity diagram describing the behavior of the invoked collaboration. A complete choreography path (i.e. starting at the initial node and ending at one or more final nodes) can then be described as a directed acyclic graph $p = (N, A_n \cup A_i)$ with the following characteristics:

a) The nodes (*N*) are CallBehaviorActions.

b) The directed arcs (both normal, A_n, and interrupting ones, A_i) represent *immediate precedence* relations, so that, given two CallBehaviorActions *n* and *m*,

   i. if a *normal* directed arc links *n* to *m*, then $n \rightsquigarrow m$ in the choreography graph.

   ii. if an *interrupting* directed arc links *n* to *m*, then *n* is inside an interruptible region *R* in the choreography graph and $u \rightsquigarrow m$, where *u* is the source node of the interrupting edge that leaves *R*.

c) If a node *n* has no incoming arcs, then $v_0 \rightsquigarrow n$ in the choreography graph. We say *n* is an initial node of the path. If a node *n* has no outgoing arcs, then $n \rightsquigarrow m$, with $m \in V_F$, in the choreography graph. We say *n* is a final node of the path.

Fork nodes describe concurrent execution of sub-collaborations in a choreography and thus create branching in paths. Join nodes, on the contrary, join several branches back into a single one. This is depicted in Figure 1(b), which shows the directed graph of one of the possible paths through the choreography in Figure 1(a). Decision nodes and interrupting edges give rise to alternative paths. Figure 1(c), for example, shows the path through the choreography in Figure 1(a) that results from traversing the interrupting edge (i.e. when the interruption takes place). We note also that loops in a choreography give rise to a number of alternative paths, each of them showing the execution of a different number of iterations.

Each choreography path defines one or more global visual orders of events. To construct each global visual order, we first select one visual order for each CallBehaviorAction in the path (note that a CallBehaviorAction may invoke a collaboration with several visual orders associated). Thereafter we compose in weak sequence the visual orders of each pair of CallBehaviorActions that are interconnected by an arc in the path. Finally, we obtain the transitive closure of all the composite visual orders. This process is formally described by the `BuildVisualOrder` algorithm (see below). We note that in the case of CallBehaviorActions with outgoing interrupting arcs, we need to consider the prefixes of their associated visual orders (see function *vo(n)*).

```
Algorithm BuildVisualOrder
  Ψ ← ∅ //Set of visual orders
  foreach initial node n_0
    foreach (E_0, <_0) ∈ vo(n_0)
      ProcessArc(n_0, E_0, E_0, <_0)
    end
  end
end

Procedure ProcessArc(n, E_n, E_p, <_p)
  foreach (n, m) ∈ A_n ∪ A_i
      foreach (E_m, <_m) ∈ vo(m)
        E ← E_p ∪ E_m
        < ← <_p ∪ <_m ∪ {(e_n, e_m) ∈ E_n × E_m : φ_n(e_n) = φ_m(e_m)}  //weak sequencing
        ProcessArc(m, E_m, E, <)
      end
  end
  if n is a final node then Ψ ← Ψ ∪ {(E_p, <_p^*)} //Obtain the transitive closure
end
```

Function $vo(n) = \begin{cases} \Lambda_n & \text{, if it does not exist } (n,m) \in A_i \\ pre(\Lambda_n), & \text{otherwise} \end{cases}$

Given a choreography *Ch*, a path *p* through *Ch*, and a global visual order $(E,<)$ defined by *p*, together with a mapping $\lambda : E \rightarrow \Sigma$ associating each event with a communication action (see Section 2), we say that a word $\omega = \lambda(e_1)\lambda(e_2)\ldots\lambda(e_{|E|})$ over the alphabet Σ^* is a *trace* of *Ch* iff $e_i < e_j$ implies $i < j$, and $e_i \neq e_j$ for $i \neq j$. We let $\mathcal{L}(Ch)$ denote the set of all traces of a choreography *Ch*.

4 Directly realized system

We consider a very straightforward approach to the realization of a distributed design from a choreography, which we call **direct realization**. The direct realization assumes that there is one system component for each composite role defined in the choreography, whose dynamic behavior is modeled in terms of message receptions and message sendings, and the order in which these actions may occur. The dynamic behavior $\mathcal{A}_p$ of each component or role *p* is described by an activity diagram, which we assume has a single FIFO input buffer $b \in M^*$ where messages received from all sources are kept until they are processed. This activity diagram is obtained by projecting the choreography onto the role played by the component (i.e. removing any action executed by other roles). No extra coordination messages or attributes are added during the projection. The set of system components, or role behaviors, obtained by direct realization of a choreography *Ch* is called **directly realized system,** and it is denoted as $\mathcal{A}_{Ch} = (\mathcal{A}_p)_{p \in R}$, where *R* is the set of roles involved in the choreography. We

assume the realized system components communicate by message passing over error-free channels.

To construct the activity diagram $\mathcal{A}_p$ for a role p, we first obtain a flattened choreography by eliminating those CallBehaviorActions that invoke a collaboration behavior defined by an activity diagram; and replacing each of these actions by the activity diagram representing the behavior of the called collaboration, assuming that there are no recursive calls. We then project the flattened choreography onto role p as follows:

- each CallBehaviorAction a where p does not participate is eliminated (i.e. a new activity edge is added from each node directly leading to a to each other node that can be directly reached form a; the a node and all its incoming and outgoing edges are then removed)
- each CallBehaviorAction that invokes a collaboration defined by a sequence diagram *SD* where p participates is replaced by a set of interconnected SendSignalActions and AcceptEventActions representing the sequence of local actions performed by p in *SD*.

The syntax of each $\mathcal{A}_p$ is similar to the syntax of choreography graphs, except that: V_{A} is no longer a set of CallBehaviorActions, but a set of SendSignalActions and AcceptEventActions; and Ω is a set of labels of the form $p!q(m)$ and $p?q(m)$. Its semantics is based on token flow. At any point in time, one or more tokens are present in an activity diagram. An action may begin execution as soon as it receives a token, provided that any other execution condition is satisfied (e.g. an AcceptEventAction can only be executed if an appropriate message reaches the first position of the input buffer). When the action completes execution, a token is offered to its outgoing edge. Fork nodes split a flow into multiple concurrent flows, so that when a fork node receives a token, copies of the token are created and offered to each of the fork's outgoing edges. Join nodes synchronize multiple flows by offering a token on their outgoing edge whenever tokens are offered on all their incoming edges. Decision nodes offer the same token to all their outgoing edges, but only one of the target nodes may accept it (if several target nodes may be able to accept the offered token, a non deterministic choice is made). Finally, all tokens and behaviors inside an interruptible region are terminated whenever a token leaves the region via an interrupting edge.

From the explanation of the token-game semantics we conclude that for each $\mathcal{A}_p$ there exists, at any point in time, a set Δ of *enabled actions* (i.e. actions that are being offered a token) and may in principle be executed. Some of these actions may be mutually exclusive, such that if one of them is eventually executed, the other ones could no longer be executed. Mutually exclusive actions are those that are offered the same token by a decision node (or a chain of decision nodes). For an action $a \in \Delta$, we denote by $MExc(a) \subset \Delta$ the set of enabled actions that are mutually exclusive with a. More formally, we define $MExc(a) = \{a' : a' \neq a, a' \notin Par(a), d \rightsquigarrow a', d \rightsquigarrow \mathrm{a}, d \in N_{\mathrm{D}}\}$, where $Par(a)$ is the set of actions specified to be executed concurrently with a (i.e. actions that appear in any path of $\mathcal{A}_p$ that contains a, but in different branches of the path than a). Some of the enabled actions may be are so-called interrupting actions, so that if they are eventually executed, other actions would cease being enabled. Interrupting actions are those that can be reached via an interrupting edge without traversing any other action node. For an interrupting action $a \in \Delta$, we denote by $Int(a) \subset \Delta$ the set of enabled actions that a would interrupt if executed. More formally, we define $Int(a) = \{a' \in \Delta : R \triangleright a', R \triangleright n, (n \rightarrow_{\mathrm{i}} a \vee n \rightarrow_{\mathrm{i}} m \rightsquigarrow a), n \in V_{\mathrm{A}}, R \in I\}$ (here we assume that the interrupting edge's source node (n) is an AcceptEventAction that is executed when a given external event happens).

A snapshot of the execution of an activity diagram $\mathcal{A}_p$ at a given point in time is called a *configuration*, and is described by a tuple $c = (\Delta, b, T)$ consisting of the set Δ of enabled actions, the content b of the input buffer and the set T of states of the activity's join nodes at the given point in time. The state $t_v \in T$ of a join node v is the number of tokens that v has received on its incoming edges since the last time a token was offered to its outgoing edge.

For an $\mathcal{A}_p$, the transition from a configuration $c = (\Delta, b, T)$ to a configuration $c' = (\Delta', b', T')$ upon the execution of an action a, written $c \xrightarrow{a} c'$, is possible if $a \in \Delta$ and either

a) a is a SendSignalAction, or

b) a is an AcceptEventAction with $\mu(a) = p?q(m)$, and $b = \omega m$ (i.e. message m is the first in the input buffer)

In the new configuration $c' = (\Delta', b', T')$, some of the actions that were enabled in c may no longer be it, while new actions may become enabled. The actions that are no longer enabled are (see (1) below), in addition to a itself, the actions interrupted by a (if a was an interrupting action), the actions that are mutually exclusive with a, and any interrupting action associated with an interruptible region that no longer contains enabled actions after the execution of a (note that for such an interrupting action b, we would have $Int(b) = \{a\}$ when a was executed). The new actions that become enabled are the actions that receive the token that is placed on a´s outgoing edge after its execution (see (2) and (4) below and note that if a leads through some path to a join node u that has not yet received tokens on all its other incoming edges, which is controlled by t_u, then u keeps the offered token and no action is enabled in that path), and if any of those actions are inside interruptible regions, the interrupting actions associated with those interruptible regions (see (3) below). The new set Δ' of enabled actions is formally defined as:

$$\Delta' = \Delta - \{a\} - Int(a) - MExc(a) - \{b : Int(b) = \{a\}\} \quad (1)$$

$$\cup\ GetTokenAfterNode(a) \cap V_\mathrm{A} \quad (2)$$

$$\cup \{b : (n \rightarrow_\mathrm{i} b \vee n \rightarrow_\mathrm{i} m \in \rightsquigarrow b), R \rhd n, R \rhd c, c \in GetTokenAfterNode(a) \cap V_\mathrm{A}, R \in I\} \quad (3)$$

where

$$GetTokenAfterNode(a) = \quad (4)$$
$$\{v_1, \ldots, v_n : a \rightarrow v_1 \rightarrow \ldots \rightarrow v_n, (v_n \in V_\mathrm{A} \vee (v_n \in V_\mathrm{J} \wedge t_{v_n} < |\{x : x \rightarrow v_i\}| - 1)),$$
$$(v_i \notin V_\mathrm{A} \wedge (v_i \notin V_\mathrm{J} \vee (v_i \in V_\mathrm{J} \wedge t_{v_i} = |\{x : x \rightarrow v_i\}| - 1)), 1 \le i < n)\}$$

The new input buffer in c' is defined as:

$$b' = \begin{cases} b, & \text{if } a \text{ is a SendSignalAction} \\ \omega, & \text{if } a \text{ is an AcceptEventAction with } \mu(a) = p?q(m) \text{ and } b = \omega m \end{cases}$$

and the state $t'_v \in T'$ of each join node $v \in V_\mathrm{J}$ in c' is defined as:

$$t'_v = \begin{cases} (t_v + 1) \bmod |\{x : x \rightarrow v\}|, & \text{if } v \in V_\mathrm{J} \cap GetTokenAfterNode(a), t_v \in T \\ t_v \in T, & \text{otherwise} \end{cases}$$

A global configuration C of a system of role activity diagrams $\mathcal{A}_{Ch} = (\mathcal{A}_p)_{p \in R}$ is the set of all local configurations at a given point in time, that is, $C = \{c_p : p \in R\}$. A transition from a global configuration $C = \{c_p : p \in R\}$ to a global configuration $C' = \{c'_p : p \in R\}$ upon the execution of an action a, written $C \xrightarrow{a} C'$, is possible if $\exists c_p = (\Delta_p, B_p, T_p)$ and $\exists c'_p = (\Delta'_p, B'_p, T'_p)$ such that $c \xrightarrow{a} c'_p$, and

(i) if a is a SendSignalAction with $\mu(a) = p!q(m)$, then $c'_q = (\Delta_q, mb_q, T_q)$ and $c'_r = c_r$ for $r \neq p$ and $r \neq q$;

(ii) if a is an AcceptEventAction with $\mu(a) = p?q(m)$, then $c'_r = c_r$ for $r \neq p$.

An *execution* of $\mathcal{A}_{Ch}$ is a sequence of global transitions $C_0 \xrightarrow{a_1} C_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} C_n$, where $C_0 = \{(GetTokenAfterNode(v_0^p) \cap V_A^p, \varepsilon, \{0 : v \in V_J^p\}) : p \in R\}$ is the *global initial configuration* (i.e. a configuration where each role activity diagram has enabled only the actions that can be reached from its initial node, has its input buffer empty and the state of each of its join nodes is set to zero). An execution generates a *trace* $a_1 a_2 \ldots a_n$. We say an execution is *successful* if it ends in the *global final configuration* $C_f = \{(\varnothing, \varepsilon, \{0 : v \in V_J^p\}) : p \in R\}$ (i.e. a configuration where each role activity diagram has no enabled actions, has its input buffer empty and the state of each of its join nodes is set to zero). A successful execution generates a *complete trace*. The set of complete traces of all successful executions of $\mathcal{A}_{Ch}$ is denoted by $\mathcal{L}(\mathcal{A}_{Ch})$.

5 Directly realizability

Definition. A choreography Ch is **directly realizable** if the following two conditions are met:

(i) The set of complete traces generated by the directly realized system is equal to the set of complete traces defined by the choreography, that is, $\mathcal{L}(\mathcal{A}_{Ch}) = \mathcal{L}(Ch)$.

(ii) Each trace generated by the directly realized system can always be extended to a complete trace.

The second condition implies that from any reachable system configuration (which was reached by a particular execution sequence) a global final configuration can be reached (through the extended complete trace). We say then that the system is **stuck-free** (see also [1]and [2]).

6 Propositions

We present now some propositions regarding the realizability of sequential compositions of collaborations. Before that, we introduce some definitions.

A role is an **initiating role** of a collaboration C if it takes the initiative to start the collaboration (i.e. the first local action it performs within C is not preceded by any other local action within C (by any other role)). The terminating roles are defined similarly. In the following, we will say that a composite-role of a collaboration is the initiating (resp. terminating) role of a sub-collaboration if it is bound to the initiating (resp. terminating) role of that sub-collaboration.

Definition 2 (weak-causality). *A weak sequential composition of two collaborations,* $C_1 \circ_w C_2$, *is* weakly-causal *if each initiating composite role of* C_2 *participates in* C_1.

Definition 3 (send-causal composition). A composition $C_1 \circ_w C_2$ *is* send-causal *if the composite role initiating* C_2 *is either the terminating role of* C_1 *or the role that performs the last sending event of* C_1.

Proposition 1. *A strong sequential composition of two directly realizable collaborations,* $C_1 \circ_s C_2$, *is directly realizable iff it is* localized*, that is, if all terminating actions of* C_1 *and all initiating actions of* C_2 *are performed by the same composite role.*

Proof. (⇐) In order to prove that the composition is directly realizable, we need to prove that (1) $\mathcal{A}_{Ch}$ is stuck-free, and (2) $\mathcal{L}(Ch) = \mathcal{L}(\mathcal{A}_{Ch})$. We prove these two conditions with help of the following property, which is always true if the sequential composition is localized: (**P**) each role is completely finished in C_1 before it sends or receives any message in C_2.

Proving that $\mathcal{A}_{Ch}$ is stuck-free is straightforward. Since C_1 and C_2 are both directly realizable, a role activity diagram may only get stuck if either there is a race and it receives a message from C_2, while still participating in C_1, or if it receives an orphan message from C_1 while already participating in C_2. However, due to **P**, neither of these cases is possible, which proves (1).

Proving that $\mathcal{L}(Ch) = \mathcal{L}(\mathcal{A}_{Ch})$ is the same as proving that $\mathcal{L}(Ch) \subseteq \mathcal{L}(\mathcal{A}_{Ch})$ and $\mathcal{L}(\mathcal{A}_{Ch}) \subseteq \mathcal{L}(Ch)$. The former is simple to see, as a result of the direct realization algorithm. To prove that $\mathcal{L}(\mathcal{A}_{Ch}) \subseteq \mathcal{L}(Ch)$ let us suppose that $\mathcal{L}(\mathcal{A}_{Ch}) \nsubseteq \mathcal{L}(Ch)$ and show that it leads to a contradiction. If $\mathcal{L}(\mathcal{A}_{Ch}) \nsubseteq \mathcal{L}(Ch)$, then $\exists \omega = e_1 \ldots e_k \in \mathcal{L}(\mathcal{A}_{Ch})$ and $\omega \notin \mathcal{L}(Ch)$. The latter might only be possible if, for any $i < j$, either (a) $e_i, e_j \in E_1 \wedge e_i \nless_1 e_j$; (b) $e_i, e_j \in E_2 \wedge e_i \nless_2 e_j$; or (c) $e_i \in E_2, e_j \in E_1 \wedge \phi(e_i) = \phi(e_j)$. Since C_1 and C_2 are directly realizable, neither (a) nor (b) can be true. And due to **P1**, (c) cannot either be true. We reach a contradiction, so $\mathcal{L}(\mathcal{A}_{Ch}) \subseteq \mathcal{L}(Ch)$. This proves (2).

Proving the other direction of the clause (⇒) is easy by contradiction assuming that the composition is not localized. (*End of proof*)

Proposition 2. *A weakly-causal sequential composition of two directly realizable collaborations,* $C_1 \circ_w C_2$, *is directly realizable if no composite role participating in* C_1 *participates with a non-initiating role in* C_2.

Proof. We just need to prove that (**P**) each role is completely finished in C_1 before it sends or receives any message in C_2. Then, following the same reasoning used in the proof of Proposition 1, we can easily prove that the direct realization of the composition is stuck-free and it generates a set of complete traces equal to the ones defined by the composition.

To prove **P** we need to consider two possible cases for a role *p*: (1) it does not participate in C_1; and (2) it participates in both C_1 and C_2. Case (1) is trivial. In case (2), according to the proposition's text, *p* is an initiating role in C_2 and will therefore always begin its participation in that collaboration with a message sending *s*. The direct realization algorithm,

and the direct realizability of C_1, ensure that *s* will never happen until *p* has executed all actions in C_1. Now, since C_2 is directly realizable, any other actions performed by *p* in C_2 will always happen after *s*, and will therefore always happen after *p* is finished with C_1. This proves **P**. (*End of proof*)

Proposition 4. *A send-causal sequential composition of two directly realizable collaborations,* $C_1 \circ_w C_2$*, is directly realizable*

- *over a communication service with* in-order delivery *if whenever a composite role plays a terminating role in* C_1 *and a non-initiating role in* C_2*, then the last message it receives in* C_1 *and the first one it receives in* C_2 *are sent by the same peer-composite role; or*
- *over a communication service with* out-of-order delivery *only if no composite role plays a terminating sub-role in* C_1 *and a non-initiating sub-role in* C_2*.*

Proof. We just need to prove that (**P**) each role is completely finished in C_1 before it sends or receives any message in C_2. Then, following the same reasoning used in the proof of Proposition 1, we can easily prove that the direct realization of the composition is stuck-free and it generates a set of complete traces equal to the ones defined by the composition.
To prove **P** we need to consider only two cases: (1) a role *p* plays a non-terminating sub-role in C_1 (i.e. ends with a sending) and a non-initiating sub-role in C_2; and (2) a role *p* plays a terminating sub-role in C_1 and a non-initiating sub-role in C_2. The other cases are covered by Proposition 3 (since send-causality implies weak-causality). For case (1) we note that, since the sequential composition is send-causal and C_1 is directly realizable, C_1 must be send-causal (otherwise either the composition could not be send-causal or C_1 would not be directly realizable). Then, the last sending by *p* in C_1 will always happen before the first sending in C_2 and, thus, before the first reception by *p* in C_2, and before any other action by *p* in C_2 (due to direct realizability of C_2). For case (2), the condition in the proposition ensures that there is no race between the last reception by *p* in C_1 and its first reception in C_2; and given that C_1 and C_2 are directly realizable, this guarantees that *p* is finished with C_1 before it sends or receives any message in C_2. This proves **P**. (*End of proof*)

References

[1] C. Fournet, T. Hoare, S. K. Rajamani, and J. Rehof, "Stuck-free Conformance", *Procs. 16th Intl. Conf on Computer Aided Verification (CAV'04)*, LNCS, vol. 3114, Springer, 2004

[2] A. Mousavi et al., "Strong safe realizability of message sequence chart specifications", Proc. Intl. Symp. on Fundamentals of Software Engineering, Springer, LNCS 4767, 2007