SOCA (2010) 4:109–136
DOI 10.1007/s11761-010-0060-9

ORIGINAL RESEARCH PAPER

Protocol contracts with application to choreographed multiparty collaborations

Ashley McNeile

Received: 15 April 2009 / Revised: 13 April 2010 / Accepted: 3 May 2010 / Published online: 16 May 2010

Abstract E-commerce collaborations and cross-organizational workflow applications are increasingly attractive given the universal connectivity provided by the Internet. Such applications are inherently concurrent and non-deterministic, so standard software engineering practices are inadequate, and we need new techniques to design extended collaborations and ensure that the implemented designs will behave correctly. The emerging technique for achieving this is to use a *choreography*, a global description of the possible sequencing of message exchange between the participants, as the basis for both the design of the collaboration and verification of its behavior. We describe a new technique that uses compositions of partial descriptions to define a choreography and show how the technique can be used to model the use of data and computation in the rules of the collaboration. We define conditions for correctness and show that they can be applied separately to each partial description. We demonstrate the expressive power of the technique with examples and discuss how it improves on previously published approaches.

Keywords Behavior contract · Verification · Service collaboration · Service choreography · Process algebra

1 Introduction

With networks now providing universal connectivity across organizational boundaries, business-to-business e-commerce and cross-organizational workflow applications are increasingly common. In such applications, two or more organizations arrange for their systems to collaborate in a shared business process, communicating by message exchange across a network infrastructure. The interaction between the systems engaged in the services can become complex, involving multiple message types and many different possible patterns of message exchange.

Current software engineering practice does not offer a generally accepted approach to the design of extended multiparty collaborations and consequently industry practice generally resorts to two expedients:

- The use of multiple simple collaborations, each involving only a pair of participants and invoking manual intervention when the collaboration enters a state that, because of the fragmentation of the process, the software is not designed to handle. While this ensures that the complexity of the software is bounded to a level that can be validated using conventional software engineering techniques, in general it results in business processes that are slower, less scalable and more expensive than they need be.
- The use of "packaged" solutions that require business processes to adhere to predefined proprietary templates. The templates are provided by specialist vendors who are able to invest in evolving and validating standard solutions over time. This works for some well-standardized applications but sometimes requires that processes are forced to fit and, more importantly, does not address cases where competitive advantage stems from innovative custom solutions.

This attests to the value in obtaining a general approach to designing extended collaborations, simple and practical enough to be used routinely in the context of custom solution design.

A. McNeile (✉)
Metamaxim Ltd., London, UK
e-mail: ashley.mcneile@metamaxim.com

1.1 The challenge

Designing services that can engage successfully in extended collaborations is a challenge. Unlike a program or process that follows a single procedural description, a collaboration involves the concurrent interaction of independent participants and is subject to various unpredictable factors at execution time that can affect its behavior, progress and outcome; in particular:

- Differences in the relative speed or scheduling of the hardware/software at each participant.
- Variations in latency of message transfer by asynchronous messaging systems.
- Execution choices made by the infrastructure, independently of the application, when handling synchronization across distributed components.[1]

Considered as a whole, the behavior of a collaboration is the emergent behavior of the interaction between its participants and is non-deterministic in nature. This non-determinism renders conventional testing techniques ineffective, as it is very hard to design a test strategy that provides adequate coverage of the execution possibilities and it is not generally possible to repeat runs to recreate errors. As Owicki and Lamport observe: "There is a rather large body of sad experience to indicate that a concurrent program can withstand very careful scrutiny without revealing its errors. The only way we can be sure that a concurrent program does what we think it does is to prove rigorously that it does it." [16].

The emerging approach to building extended service collaborations that are provably reliable is to base the design on a *global contract* that specifies the allowable patterns of message exchange between the participants from a global perspective. Such a global contract is known as a *choreography*. Building a collaboration based on a choreography has three steps as depicted in Fig. 1:

1. At design-time, a choreography is developed which describes all meaningful sequences of message exchange between the participants.
2. Using some kind of mechanical process, a specification of the behavior of each participant is extracted from the overall choreography.[2]
3. At run time, the participants interact, each behaving according to its own behavior specification.

[1] Such as using a 2-phase commit mechanism for synchronous message exchange.

[2] In the jargon of choreography theory this is called "end-point projection".

At run time (step 3), the participants behave independently. Each is free to send and receive messages according to its own behavioral specification and without any central orchestrating or controlling entity to enforce conformance to the original choreography.

This procedure guarantees a degree of compatibility between the designs of the behaviors of the participants but, unfortunately, this is not enough by itself to guarantee that the behavior of the collaboration at step 3 does not depart from the choreography. It may be that the collaboration deadlocks leaving messages unprocessed or allows patterns of message exchange not envisaged in the original choreography definition and which result in combinations of states of the participants that were not intended and have no meaning. As the choreography specifies the meaningful sequences of exchange, it is clearly important that departure from it is not possible, as such departure represents entering an unintended and meaningless state. The behavioral rules embedded in the participants must therefore be strong enough to prevent such departure and this, in turn, requires that the choreography conforms to certain structural conditions. The problem of determining general conditions on the form a choreography which are sufficient to guarantee that the interaction of extracted behaviors is **bound to adhere** to the choreography is known as *the realizability problem*.

1.2 Choreography realizability

Realizability of a choreography is determined by a combination of:

- The *communication model* used for message exchange between participants. This model describes the communication channels (or queues) between the participants and is characterized by the number of the queues, message ordering discipline of the queues, queue size bounds etc.
- Conditions on the form of the choreography itself.
- The mechanical process used to extract the participant behavior specifications from it.

Investigations into choreography realizability address the question: *Given a particular communication model* (*e.g., unbounded FIFO queues in each direction between each pair of participants*)*, and a process for extracting participant behaviors from a choreography, what are sufficient conditions on the form of the choreography to guarantee that the collaboration will always succeed?* This is the key theoretical research challenge in choreography theory, and the focus of this paper is an investigation into this question. The idea is that if sufficiently simple and robust conditions on the form a choreography can be formulated, along with

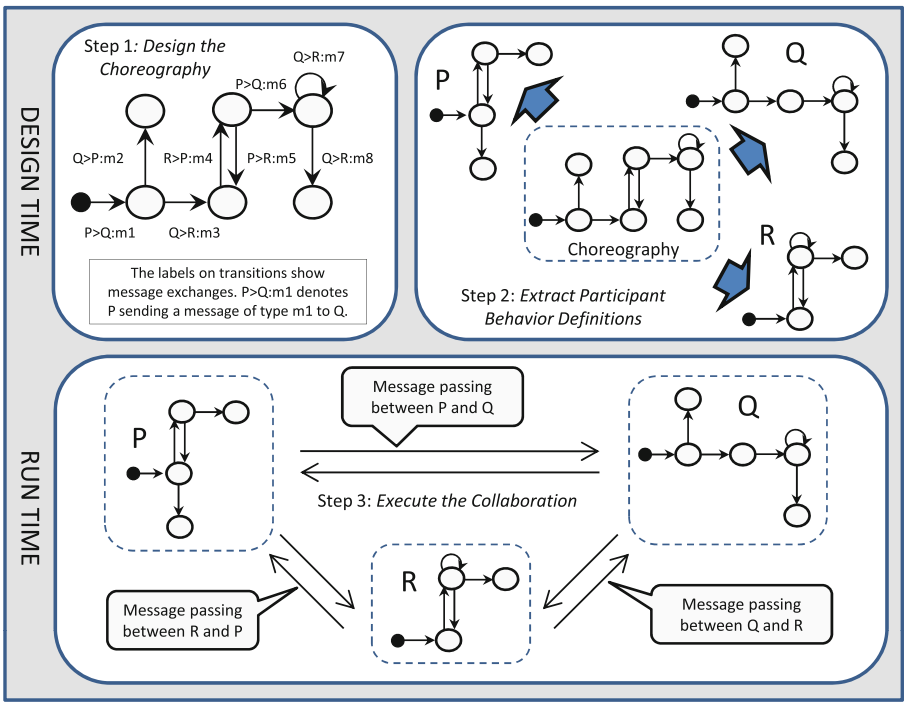

Fig. 1 Extracting participant designs from a choreography

a simple procedure for extracting participant behavior definitions, then designing extended collaborations can become routine. If the choreography is built to obey the rules, the resulting collaboration is bound to work.

1.3 Contribution

Over the last few years, a number of approaches to the realization problem have been proposed: some of the main ones are discussed toward the end of this paper in Sect. 11. We claim that the approach we define in this paper makes a significant advance over other proposals in three respects:

- We use *composition* in the articulation of a choreography, so that a single choreography can be expressed as a composition of partially synchronized descriptions. This gives our approach the expressive power to model cases where the ordering and/or certainty of message receipt is unknown in advance and cannot be constrained by the choreography. These kinds of collaboration are hard or impossible to handle without composition.
- We demonstrate that realizability of a choreography defined as a composition only needs to be established *individually for the components of the composition*. Determining realizability for a single component is easier than analyzing the choreography as a whole, thus simplifying the analysis.
- We allow the use of *data and computation* in the definition of a choreography, so that the decision on whether it is permissable to engage in a message exchange can be determined by the result of a calculation. This is in contrast to most other published approaches, which only guarantee realizability on the basis of patterns of message exchange.

1.4 Organization of this paper

The paper is organized as follows:

Section 2 provides background on *Protocol Modeling* (PM) whose formalisms provide the basis for the rest of the paper.

Section 3 introduces the concept of a *protocol contract* together with how its satisfaction is defined and established. **Section 4** discusses how protocol contracts can be composed and decomposed.

Section 5 extends the basic PM and protocol contract formalisms to describe choreographies and participant contracts.

Sections 6, 7, 8 and **9** investigate the question of realizability and give sufficient conditions for realizability in both synchronous and asynchronous collaborations.

Section 10 gives an example of a three party collaboration, showing the projected participant contracts for asynchronous collaboration.

Section 11 relates our work to other research into choreography realization and **Sect. 12** concludes with our view of the significance of this work.

2 Protocol modeling

Protocol Modeling (PM) is a technique that aims to combine the strengths of process algebra with the ability to model data. PM borrows from process algebra in establishing a formal relationship between the states of a process and the events it can handle, and in supporting process composition. It departs from traditional process algebra in allowing processes to maintain data and allowing states to be computed from stored data. The following small example introduces PM and illustrates how a PM representation relates to pure algebraic representation. This account of PM is brief and intended to provide a basis for the rest of this paper; and a complete account of can be found in [10].

2.1 Protocol model example

Consider the model shown in Fig. 2. The left-hand side (the "algebraic" description) shows a state machine for a very simple (and limited) bank account. The account can have a balance of $\{-2, -1, 0, 1, 2\}$ represented by states $\{B{-}2, B{-}1, B0, B1, B2\}$. It has an alphabet $\{Open, D1, D2, W1, W2, Close\}$ of events that it understands, where $D1, W1$ deposit and withdraw one unit respectively and $D2, W2$ deposit and withdraw two units respectively. The left-hand side process can be described in algebraic form (using Milner's CCS [14] notation) as follows:

$$
\begin{aligned}
&ACCOUNT = Open.B0\\
&B0 = D1.B1 + D2.B2 + W1.B{-}1\\
&\quad + W2.B{-}2 + Close.Closed\\
&B1 = D1.B2 + W1.B0 + W2.B{-}1 + Close.Closed\\
&B2 = W1.B1 + W2.B0 + Close.Closed\\
&B{-}1 = D1.B0 + D2.B1 + W1.B{-}2\\
&B{-}2 = D1.B{-}1 + D2.B0
\end{aligned}
$$

The right-hand side of Fig. 2 shows the same bank account rendered in PM notation. The PM description comprises three machines, $A1$, $A2$ and $A3$. These are called *protocol machines*.

- $A1$ provides the basic account behavior, showing that the $Open$ must happen first, followed by any number of deposits $\{D1, D2\}$ and withdraws $\{W1, W2\}$, followed by the $Close$. This machine maintains a $balance$ attribute for the account using the updates indicated in the bubbles attached to the transitions. The $balance$ is a numeric attribute owned by $A1$ and only $A1$ may alter its value, although other composed machines may access it. The current state is also an attribute of the machine, with an enumerated type allowing values $\{dot, active, closed\}$.
- $A2$ expresses the fact that the account may only be closed if it is in credit. Instead of having its state stored as part of the local storage, this machine has its state represented by a *state function* (shown in the $A2$ cartouche in the diagram) that returns an enumerated type, allowing values $\{in\ credit, overdrawn\}$. We describe this as a machine with a *derived state*. Because their states are derived rather than driven by transitions, protocol machines that use a derived state can be topologically disconnected. For graphical emphasis, the state icons in a machine with a derived state are shown with a double outline.
- $A3$ expresses the fact that the balance is limited to the range -2 to $+2$. The machine does this by constraining the deposit and withdraw events on the basis that their *ending* states must be in this range. In PM, this is called a *post-state constraint*. A machine is unable to accept an event that violates a post-state constraint, so in CSP terms it is a *refusal* exactly as if there was no outgoing transition for the event from the current state of the machine.

These three machines are in parallel composition using the parallel composition operator of Hoare's CSP [5], so the behavior of the bank account is given by $A1 \parallel A2 \parallel A3$. This composition is also a protocol machine and has a behavior that is indistinguishable, by black-box testing, from the algebraic description. For instance, the *Close* can only take place if $A1$ is in state *active* and $A2$ is in state *in credit*, mirroring the fact that the left-hand diagram only has outgoing transitions for *Close* from the states $\{B0, B1, B2\}$.

While this approach is based on parallel composition of conceptually concurrent machines, it is important to appreciate that there is no requirement or implication that multiple virtual processors or threads are used. We will assume that the assembly of composed machines, such as $A1 \parallel A2 \parallel A3$, is single threaded.[3]

Clearly, the PM model on the right-hand side of Fig. 2 can be modified to handle balances ranging $-\infty$ to $+\infty$ (assuming no limit on the ability of the $balance$ attribute to represent any numeric value) by:

- Replacing $\{D1, D2, W1, W2\}$ with generic *Deposit* and *Withdraw* events carrying an $amount$ field,[4] and using this field to update the $balance$ appropriately in $A1$.
- Removing $A3$.

Making the equivalent change to the algebraic model would require an infinite state space, as the data owned and main-

[3] The ModelScope tool [11] that supports Protocol Modeling uses a single threaded implementation of CSP ‖ composition.

[4] We use the term *field* in the context of messages and *attribute* in the context of machine local storage.

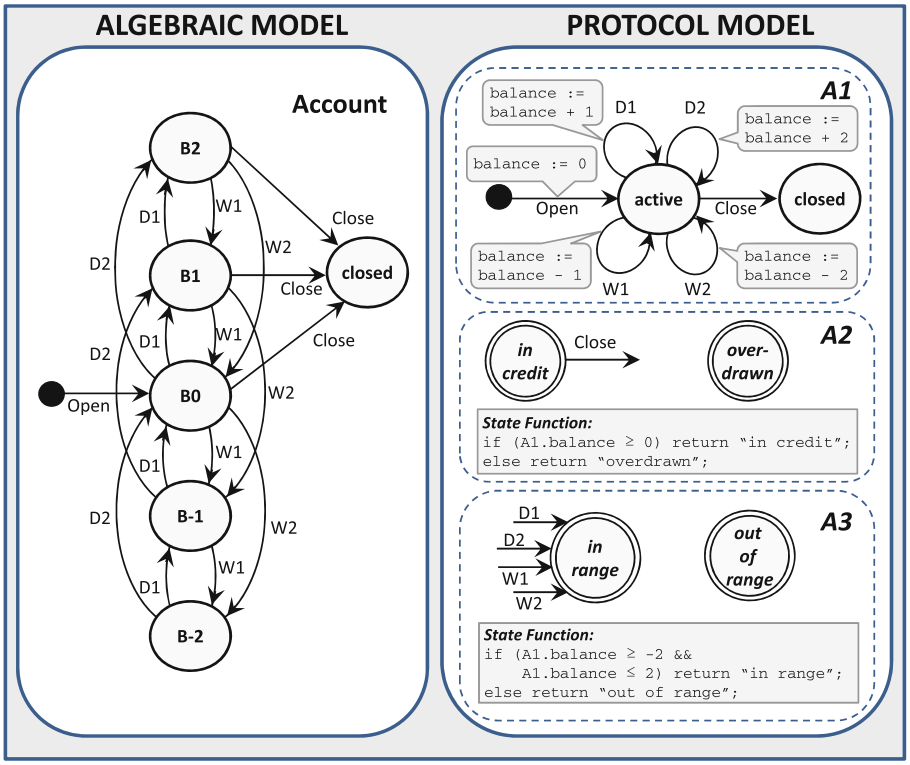

Fig. 2 Bank account as algebraic and protocol model

tained by the system has to be encoded in the state space of the process.

2.2 Terms and definitions

This section provides some further definitions and terminology related to protocol machines that are germane to the paper.

Alphabet. Every protocol machine divides the universe of possible events into those that it understands (and will either accept or refuse) and those that it does not (and ignores). This is done on the basis of the *alphabet* of a process: the set of event types for which the behavior of the process is defined. Thus, the alphabet, denoted $\alpha(A1)$, of the machine $A1$ in the example (once amended to handle arbitrary balance values) is the set $\{Open, Deposit, Withdraw, Close\}$.

Attributes. A machine owns a set of attributes, each of which is either *stored* or *derived*. Only the machine that owns a stored attribute can alter its value, and only when moving to a new state in response to an event. The updates for stored attributes are shown as bubbles attached to transitions. A derived attribute is not stored and has its value returned on demand by a derivation (calculation) function using the attributes of the owning machine and other composed machines.

Dependent and independent machines. We distinguish two types of machine:

- A *dependent* machine is one that accesses the values of attributes that it does not own (i.e., that belong to other machines composed with it). Both $A2$ and $A3$, considered as "stand-alone" machines, are dependent as they use *balance*, which they do not own, to compute their state.
- An *independent* machine is one that makes no such access. $A1$ is independent; as is the protocol machine formed as the composition of $A1$ and $A2$ as the reference to the *balance* attribute used by $A2$ to determine its state is resolved by $A1$, which is within the composition.

While derived state calculations are the source of dependency in the machines $A2$ and $A3$, the definition extends to **any** inter-machine reference whereby one machine uses the attribute of another to:

- compute the value of a derived state, or
- compute the value of a derived attribute, or
- compute the new value of a stored attribute in response to accepting an event.

Note that two machines having elements of their alphabets in common **is not** a source of dependency between them.

Thus, $A1$ is independent even though event types in its alphabet are also in the alphabets of $A2$ and $A3$.

Composition. We will use the symbol $\|$ to indicate CSP parallel composition, with the semantics defined by Hoare in [5]. We use the term *assembly* to describe a machine created by parallel composition of other machines; and we allow an attribute in one component of an assembly to resolve a reference in another, as `balance` in $A1$ resolves the references in $A2$ and $A3$.

Trace. A sequence of event instances, `t`, is a *trace* of an independent machine, M, if there is an execution of M starting from the machine's initial state that accepts `t`. Each element of a trace completely specifies both the type of the event instance and the values of all fields carried by the instance. Thus, a trace of $A1$ (once amended to handle arbitrary balance values) might be ⟨`Open`, `Deposit 100`, `Withdraw 80`, `Deposit 50`⟩. The set of traces of M is denoted by $traces(M)$.

If M is a dependent machine, `t` is trace of M iff $\exists\ X$ such that $M \parallel X$ is independent and $\mathtt{t} \in traces(M \parallel X) \restriction_{\alpha(M)}$. In other words, there is a trace of $M \parallel X$ which, if all entries that are not in the alphabet of M are removed, is equal to `t`.

Determinism. Protocol machines are deterministic, in that the set of events that a given independent machine can accept at any point is fully determined by the trace accepted by the machine up to that point. More formally: If X is an independent machine (i.e., one with no unresolved references) and two executions of X behave as follows:

- In one execution, X accepts a trace, `t`, of event instances and then accepts (refuses) a further event instance, `e++`,[5]
- In the other, X accepts `t` but then refuses (accepts) `e++`.

Then X is **not** a protocol machine.

Determinism of a dependent machine is defined by the requirement that any composition of it with another protocol machine that resolves its references must yield an independent machine that qualifies as a protocol machine according to the above.

Because the component machines are deterministic, so are assemblies. As Hoare himself notes: "…the concurrent operator by itself does not introduce non-determinism" [6].

Equivalence. A consequence of determinism is that if two independent machines have the same set of traces, no black-box experiment can determine the difference between them, and they are therefore considered to be the same machine. Two dependent machines are considered to be the same if one can be substituted for the other in any independent assembly without affecting the trace set of the assembly.

[5] We use the notation ++ throughout this paper as a visible reminder that a trace element is an *instance*, carrying values in all its fields.

2.3 Topological representation

Much of the later parts of this paper concern reasoning based on the topology of protocol machines represented as state transition diagrams. As a basis for this we define a *representation* of a stored state machine as a tuple $P = (\Lambda_P, \Sigma_P, \Gamma_P, \Delta_P)$ where:

- Λ_P is the alphabet of P, the set of events which P will either accept or refuse. So $\Lambda_P = \alpha(P)$. Elements range over $a, b, \ldots$.
- Σ_P is a finite set of states of P. Elements range over $\sigma_i, \sigma_j, \ldots$.
- $\Gamma_P \subseteq (\Lambda_P \times \Sigma_P)$ is a binary relation. $(a, \sigma_i) \in \Gamma_P$ means that a can be accepted by P when P is in state σ_i. $(a, \sigma_j) \notin \Gamma_P$ means that a is refused by P when P is in state σ_j.
- Δ_P is a total mapping $\Gamma_P \to \Sigma_P$ that defines for each member of Γ_P the new state that P adopts as a result of accepting an event. So $\Delta_P(a, \sigma_k) = \sigma_l$ means that if P accepts a when in state σ_k it will then adopt state σ_l.

Clearly, this has a direct translation into a graphical state transition diagram. Note that this kind of representation is defined for *stored state machines*, where the new state that a machine adopts as the result of accepting an event is driven by topological rules (embodied in the function Δ). It is not defined for *derived state machines* as their behavior cannot be depicted in pure topological form.

Suppose we have two stored state machines, P and Q. Using the semantics of CSP $\|$ composition we can create a representation of $P \parallel Q$ as follows:

- $\Lambda_{P\|Q} = \Lambda_P \cup \Lambda_Q$
- $\Sigma_{P\|Q} = \Sigma_P \times \Sigma_Q$ (the Cartesian product of Σ_P and Σ_Q)
- Given $a \in \Lambda_{P\|Q}$ and a state $(\sigma_p, \sigma_q) \in \Sigma_{P\|Q}$, we determine whether $(a, (\sigma_p, \sigma_q)) \in \Gamma_{P\|Q}$ as follows:
  - $(a \in \Lambda_P \wedge (a, \sigma_p) \notin \Gamma_P) \Rightarrow (a, (\sigma_p, \sigma_q)) \notin \Gamma_{P\|Q}$
  - $(a \in \Lambda_Q \wedge (a, \sigma_q) \notin \Gamma_Q) \Rightarrow (a, (\sigma_p, \sigma_q)) \notin \Gamma_{P\|Q}$
  - Otherwise, $(a, (\sigma_p, \sigma_q)) \in \Gamma_{P\|Q}$
- And we construct $\Delta_{P\|Q}(a, (\sigma_p, \sigma_q))$ as follows:
  - $(a \in \Lambda_P) \wedge (a \notin \Lambda_Q) \Rightarrow \Delta_{P\|Q}(a, (\sigma_p, \sigma_q)) = (\Delta_P(a, \sigma_p), \sigma_q)$
  - $(a \notin \Lambda_P) \wedge (a \in \Lambda_Q) \Rightarrow \Delta_{P\|Q}(a, (\sigma_p, \sigma_q)) = (\sigma_p, \Delta_Q(a, \sigma_q))$

- $(a \notin \Lambda_P) \wedge (a \notin \Lambda_Q) \Rightarrow \Delta_{P\|Q}(a, (\sigma_p, \sigma_q)) = (\sigma_p, \sigma_q)$
- Otherwise $\Delta_{P\|Q}(a, (\sigma_p, \sigma_q)) = \Delta_P(a, \sigma_p), \Delta_Q(a, \sigma_q))$

We will be using this construction in Sect. 8, in the context of choreography analysis.

3 Protocol contracts

We now introduce the concept of a *protocol contract*, which is a partial specification of the behavior of a protocol machine. Later in the paper, we use protocol contracts as the vehicle to capture required participant behaviors extracted from a choreography.

3.1 Definition of a protocol contract

A protocol contract $\mathbb{C}$ is a pair $[C, F]$ where:

- C is an independent protocol machine, called the *contract machine* of $\mathbb{C}$.
- F is a set of event types, $F \subseteq \alpha(C)$, called the *fully constrained* event types of $\mathbb{C}$.

An independent protocol machine M satisfies a contract $\mathbb{C}$, written $M \vdash \mathbb{C}$, iff:

$$M \parallel C = M \tag{3.1}$$

and:

$$\forall\, e \in F\ \exists\ X \text{ such that } M = X \parallel C \text{ and } X \text{ can never refuse an event of type } e \tag{3.2}$$

The first condition, (3.1), has two implications:

- The alphabet of M is a superset of that of C. This is natural, as you would not expect a design to satisfy a contract if its alphabet does not include all the event types in the alphabet of the contract.
- Every trace of M, when restricted by eliminating those events whose type is not in the alphabet of C, must be a trace of C. This condition is described as M being *Observationally Consistent*[6] with C.

The second condition, (3.2), requires that an X can be found so that either e is not in the alphabet of X or every state of X allows e to be accepted. If an event type is *fully constrained* by the contract, acceptability by the contract machine is a **necessary and sufficient** condition for acceptability by M.

[6] As defined by Ebert and Engels in [2].

If an event is *not fully constrained* by the contract, acceptability by the contract machine is a necessary **but not a sufficient** condition for acceptability by M. We give an example in the next section.

3.2 Protocol contract example

Consider Fig. 3. The left-hand side specifies a *contract* for the behavior of a bank account. The right-hand side shows the *design* of a bank account, $Account = A1 \parallel A2 \parallel A3$. We now show that the design satisfies the contract.

Account clearly obeys the upper part (state machine) of the contract, as the state diagram $A1$ of the Account is the same as the state diagram, C, of the contract. This means that $Account \parallel C = Account$, thus meeting (3.1).

We now wish to show that *Account* meets the lower part of the contract, which lists the events that are *fully constrained* by the contract. To do this, consider the *Account* reformulated as shown in Fig. 4 so that:

$$Account = A1' \parallel A1'' \parallel A2 \parallel A3$$

In this reformulation, the machine $A1$ has been split into two machines, $A1'$ and $A1''$. The first of these specifies the event sequencing on *Open*, *Deposit*, *Withdraw* and *Close*; and the second maintains the *balance* but does not constrain event sequencing.

The machine $A1'$ is now identical to C, so:

$$Account = C \parallel A1'' \parallel A2 \parallel A3$$

The machine $A1'' \parallel A2 \parallel A3$ plays the role of X for both *Open* and *Deposit* in part (3.2) of the contract definition above. As none of $A1''$, $A2$ or $A3$ can ever refuse the events *Open* and *Deposit*, these events are fully constrained by the contract. However, as $A2$ can refuse *Close* and $A3$ can refuse *Withdraw*, these events are **not** fully constrained by the contract.

3.3 Discussion

Suppose I have a bank account and I know, because the bank tells me, that the account conforms to the contract shown on the left of Fig. 3. Suppose also that I know that the account has been opened but not closed. Then I know that, in terms of the contract, it is in the state *active*. From this I can deduce that:

- I can do a *Deposit* and the account will accept this in its current state. This is because *Deposit* is fully constrained by the contract.
- I **may or may not** be able to *Withdraw* or *Close*, as the contract does not fully constrain these and so does not guarantee that these will be accepted when in the contract is in the active state. In other words, the bank has

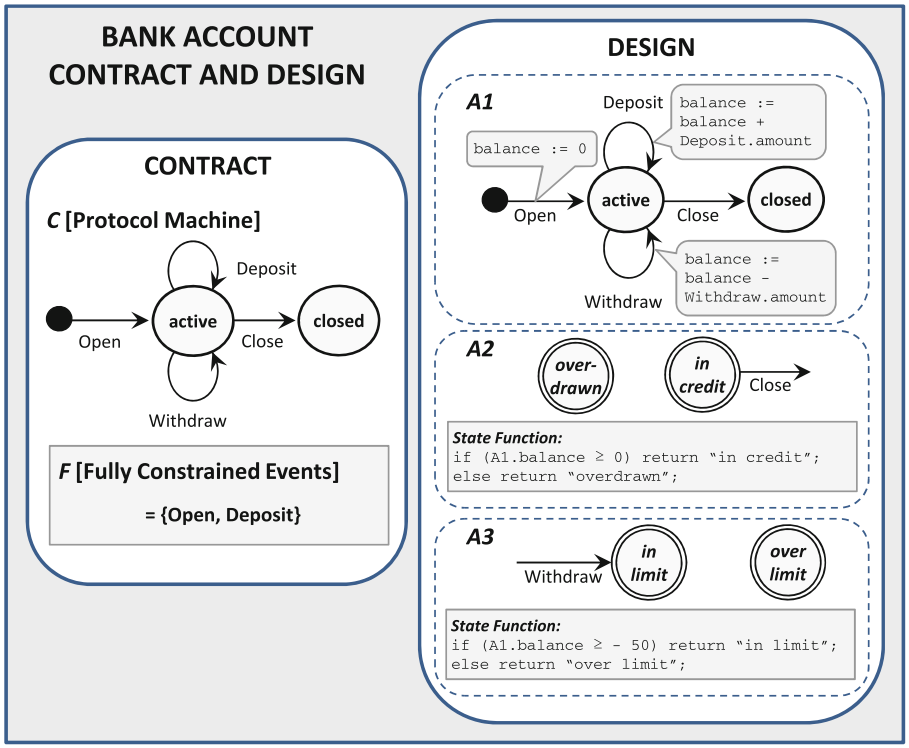

Fig. 3 Bank account: contract and design

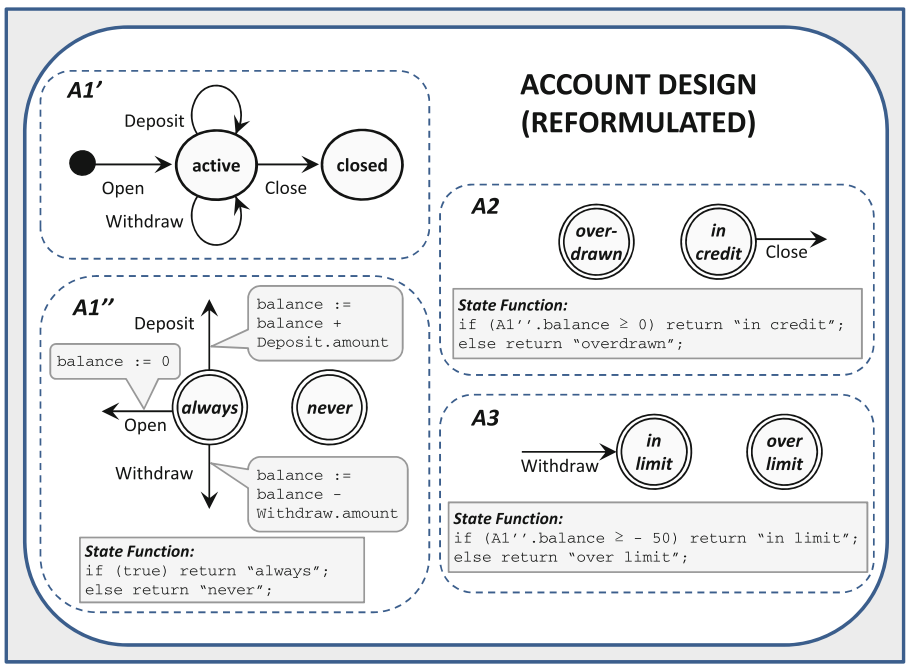

Fig. 4 Bank account reformulated

rules about when a withdraw can be made and when an account may be closed, but the contract does not include these rules.

– If I close the account, and the bank permits this operation, I cannot then do any further deposits or withdraws. This is because the sequencing rules of the contract cannot be violated.

In this way, a contract allows determination of what is possible for a machine, based on partial knowledge of its state.

3.4 Dependency in contracts

Suppose that we want to enhance the contract for the Bank Account to fully constrain *Withdraw* events, with the rule that

a withdraw must not take the account overdrawn beyond its limit. The obvious way to do this is the include the machine $A3$ in the contract, so that the protocol machine of the contract becomes $C \parallel A3$ where C is the state diagram shown on the left of Fig. 3. However, $A3$ has a derived state based on *balance*, and the contract needs to know what this is as the contract machine must be independent. This means that *balance* has to be defined within the contract and so we add it as an attribute to C, along with the update rules (as shown in the bubbles in $A1$) that maintain its value. This means that C is now identical to $A1$.

In general, we allow the use of dependent machines in the definition of a contract provided that they are in composition with other machines so that the contract as a whole is independent. This requires that any inter-machine references used in the contract are fully resolved within the contract.

4 Composition and decomposition of contracts

Protocol contracts can be composed and sometimes decomposed, as we describe below.

4.1 Composition

If $\mathbb{C}_1 = [C_1,\ F_1]$ and $\mathbb{C}_2 = [C_2,\ F_2]$, then their composition is defined as follows:

$$\mathbb{C}_1 \oplus \mathbb{C}_2 = [C_1 \parallel C_2,\ F_1 \cup F_2]$$

It is easy to show that if a machine P satisfies both $\mathbb{C}_1$ and $\mathbb{C}_2$, it also satisfies $\mathbb{C}_1 \oplus \mathbb{C}_2$:

1. $P \vdash \mathbb{C}_1 \Rightarrow P = C_1 \parallel P$
   $P \vdash \mathbb{C}_2 \Rightarrow P = C_2 \parallel P$
   so $P = (C_1 \parallel C_2) \parallel P$.

2. Suppose $e \in F_1$. Then $\exists\ X$ s.t. $P = C_1 \parallel X$ and X never refuses e.
   As $P \vdash \mathbb{C}_2$, $P = C_2 \parallel P$
   so $P = (C_2 \parallel C_1) \parallel X$.

4.2 Decomposition

Suppose that $\mathbb{C} = [C,\ F]$ and that $C = C_1 \parallel C_2$. It is reasonable to ask whether $\mathbb{C}$ can be decomposed into two contracts, $\mathbb{C}_1$ and $\mathbb{C}_2$, the former using C_1 and the latter using C_2, in such a way that:

$$P \vdash \mathbb{C} \Rightarrow (P \vdash \mathbb{C}_1) \wedge (P \vdash \mathbb{C}_2)$$

This is possible provided that:

$$F \cap \alpha(C_1) \cap \alpha(C_2) = \emptyset \tag{4.1}$$

and the decomposition, giving $\mathbb{C} = \mathbb{C}_1 \oplus \mathbb{C}_2$, is:

$$\mathbb{C}_1 = [C_1, F \cap \alpha(C_1)] \quad \text{and} \quad \mathbb{C}_2 = [C_2, F \cap \alpha(C_2)]$$

The reason for the stricture (4.1) is as follows. Suppose that $P \vdash \mathbb{C}$ and that $e \in (F \cap \alpha(C_1) \cap \alpha(C_2))$. While the pattern of occurrences of e in P is fully described (i.e., fully constrained) by $C_1 \parallel C_2$, it may not be fully described by either C_1 or C_2 alone because it depends on the way the two processes synchronize on e. In this case e cannot belong to the set of the events fully constrained by a contract defined in terms of either C_1 or C_2 alone, so the decomposition fails.

Suppose that stricture (4.1) is met and that $P \vdash \mathbb{C}$. We have:

$$P = C \parallel P = C_1 \parallel C_2 \parallel P \tag{4.2}$$

As it is a property of CSP $\parallel$ that for any A, B with $\alpha(B) \subseteq \alpha(A)$:

$$traces(A) \supseteq traces(A \parallel B) \tag{4.3}$$

we have by (4.2) and (4.3):

$$traces(P) \supseteq traces(P \parallel C_1) \supseteq traces(P \parallel C_1 \parallel C_2) = traces(P)$$

so $traces(P) = traces(C_1 \parallel P)$, and hence:

$$P = C_1 \parallel P \tag{4.4}$$

Suppose $e \in F \cap \alpha(C_1)$. As $P \vdash C$, $\exists\ X$ s.t. $P = C \parallel X$ and X never refuses e. So, $P = C_1 \parallel (C_2 \parallel X)$. As, by stricture (4.1), $e \notin \alpha(C_2)$, $X' = (C_2 \parallel X)$ can never refuse e, and we have:

$$P = C_1 \parallel X' \text{ and } X' \text{ can never refuse } e \tag{4.5}$$

(4.4) and (4.5) $\Rightarrow P \vdash \mathbb{C}_1$. Similarly, $P \vdash \mathbb{C}_2$.

Finally, combining the results for composition and decomposition we note that:

$$\text{If } F_1 \cap F_2 = \emptyset,\ (P \vdash \mathbb{C}_1 \oplus \mathbb{C}_2) \Leftrightarrow (P \vdash \mathbb{C}_1 \wedge P \vdash \mathbb{C}_2)$$

5 Choreography and collaboration

In this section, we explore the use of Protocol Modeling ideas to specify a *choreography* for a collaboration.

Our approach uses *end-point projection*, whereby the behavior of the participants is abstracted from the choreography. This is conceptually similar to projecting a WS-CDL choreography into BPEL end-point (participant) definitions, for instance as described by Mendling et al. [13]. The difference in our scheme is that, rather than projecting into any particular design language, a choreography is projected to define *protocol contracts* for the end-points. As protocol contracts can be composed, it is possible for a participant to engage in

multiple choreographies by complying simultaneously with the different contracts they impart.

5.1 Choreography definition

We define a choreography, $\mathfrak{C}$, as a *choreography protocol machine*. Instead of an alphabet of *event types* as we had in in Sect. 2, a choreography protocol machine has an alphabet of *message exchanges*. The upper cartouche in Fig. 5 shows a choreography for collaboration between two participants, a *Customer* (*Cust*) and a *Supplier* (*Supp*), in the context of order processing. The transitions are labeled with the message type being exchanged, with a prefix indicating the sender and receiver of the message. So a label:

Supp > *Cust:Accept Order*

represents the exchange of an *Accept Order* message sent by participant *Supp* to participant *Cust*. These exchanges, defined by the combination of sender, receiver and message type, are the individuals that form the alphabet of the choreography.

Roughly speaking, a choreography represents the set of possible orderings of message exchange between the participants of the collaboration; the exact definition will be given later, in Sect. 6.2. The choreography in Fig. 5, shown in the upper cartouche of the figure, is represented as two machines which are taken to be in CSP ‖ composition, so that they synchronize on message exchanges that appear in the alphabet of both but are otherwise independent. The set of possible orderings of exchanges between Customer and Supplier given by this choreography is the set of traces of the composition of these two machines. This idea, that CSP compositions are used to generate the set of possible exchanges in a collaboration, is central to the approach described in this paper.

If `t` is a sequence of entries of the form `P>Q:m++` we define `t` to be a trace of an independent choreography machine if it is accepted by the machine. Note that the machines of a choreography can have attributes and derived states, so a trace has to be defined in terms of a sequence of message *instances*, loaded with field values (as indicated by the ++ in the entry). The choreography of a collaboration is well constructed if it has a set of traces that covers all possible useful interaction scenarios. We will assume that choreographies are, in this sense, well constructed.

The statement *The Choreography Fully Constrains the Receivers* specified as part of the choreography means that it fully describes the circumstances under which a message can be received. This assumption is required to establish realizability, and we will assume, as standard, that choreographies fully constrain message reception in the participants.

5.2 Collaboration contracts

We define the behavior of the participants in our collaboration using protocol contracts. The elements of the alphabet of the contracts represent *sending* and *receiving* messages, and we refer to these as *actions*. An element of the alphabet of a contract is of the form:

- !>*Q:m*1 indicating a *send action* that sends a message of type *m*1 to participant *Q*; or
- ?<*R:m*2 indicating a *receive action* to receive a message of type *m*2 from participant *R*.

Composition of machines of a participant is CSP ‖ composition over the action alphabets of the composed machines. Accordingly, the synchronization across composed machines within a given participant is *on both send and receive actions*.

The fields of output messages are considered to be derived (calculated by a derivation function) using the attributes of the machines of a participant. This creates a dependency between the machines and the fields, whereby an output message field f is dependent on a machine M if its derivation function uses (directly or indirectly) any attribute of M. For a machine M to be independent, we require that (in addition to the conditions described earlier in Sect. 2.2):

- If f is a field of a message type output by M, either f has no dependency on M or M fully resolves all dependencies of f.

The entries of a trace are now the instances of message send and message receive actions performed by the machine in an execution. Each element of the trace fully specifies both the action in which the machine engages and the values of all fields carried by the message sent or received. If a sequence `t` of action instances performed by an independent machine M is a trace of M, then `t⌢!>P:m++`, where ⌢ means concatenation[7] and `m++` means a message instance of type *m* along with the values of all fields it carries, is also a trace iff both of the following hold:

- `t` brings M to a state (in terms of its protocol) in which it allows a transition for the action !>*P:m*
- `t` brings M to a state (in terms of its attributes) in which the derivation function for any field of *m* that depends on M returns the same value as that field has in `m++`.

We will create the contracts for participants by projection from the choreography. The lower cartouche of Fig. 5 shows

[7] So if `t = <a,b>` then `t⌢c = <a,b,c>`.

the projected contract for $Cust$. The prefixes on the transition labels in the choreography are changed in the obvious way, so in constructing the $Cust$ contract:

- a prefix "*Cust*>*Supp:*" in the choreography becomes "!>*Supp:*" in the contract; and
- a prefix "*Supp*>*Cust:*" in the choreography becomes "?<*Supp:*" in the contract.

The choreography shown in Fig. 5 involves only two participants and in this case the projection has exactly the same topology as the choreography. If there are more than two participants in a collaboration the contracts will not have exactly the same topology as the choreography, as we shall discuss in Sect. 7.

The statement "The choreography fully constrains all receive actions." shown below the choreography state diagrams is projected into the lists of fully constrained actions (the receive actions) shown in the contract as F_{Cust}.

5.3 Computations in choreographies

The example in 5 only uses state transition topology to define the choreography. But Protocol Modeling also allows the use of computation (to determine the values of attributes, fields and states), and this can be used in addition to pure topology in the definition of choreography.

A choreography may maintain attributes and use derived state machines. Where a message exchange is constrained by a derived state machine to start at a state with value `s` it has the following semantics:

- The sender can only send when in state `s`; and
- The receiver can only receive when in state `s`.

The reason for this interpretation is:

- If only the sender were constrained, the constraint should properly represented as a business rule in the sender rather than as part of the choreography.
- If only the receiver were constrained, the choreography would not in general be realizable as the receiver is never guaranteed to enter the required state, `s`.

Realizability requires that both sender and receiver be able to determine whether they are or are not in state `s`. It also

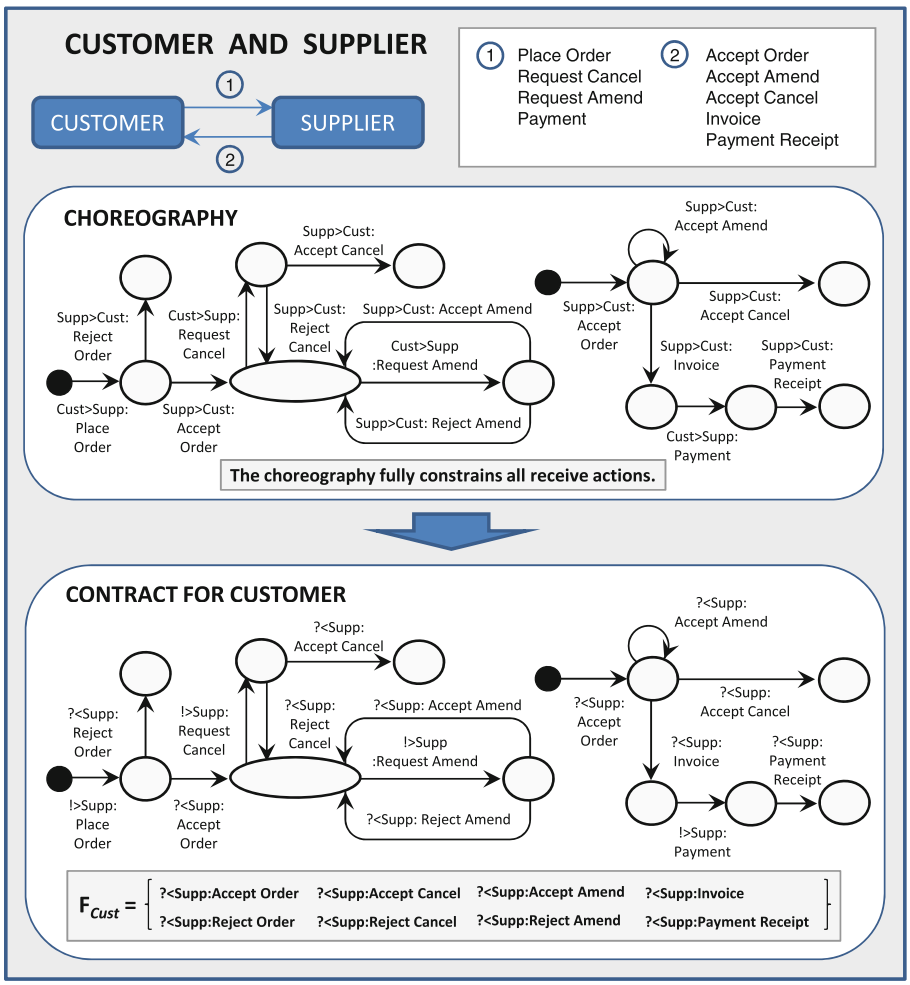

Fig. 5 A two party choreography

means that, because the sender knows both the starting state and the attributes of the message being sent, it can determine the result of the message on the state of the receiver, and this means that post-state constraints are not required in the definition of a choreography. A choreography that uses a derived state is described in Sect. 8.4.

It is also possible for a choreography to make use of derived attributes and derived fields. These features have no protocol (behavior) implications, so their use is transparent to realizability analysis. The choreography examples in this paper do not illustrate these features, but their use is straightforward.

5.4 Proof structure

Our aim is to be able to construct (design) choreographies that we **know** to be realizable, and we now embark on establishing conditions on a choreography that guarantee realizability.

As discussed by Kazhamiakin and Pistore [8], the rules for realizability depend on the form of communication model used for the collaboration, in particular whether communication between participants is synchronous or asynchronous. Our main interest is in collaborations that use *asynchronous* communication. This means that a sender transmits a message on the assumption that the receiver is able to receive it, and the message may take time to transit so that the receive happens at a later time. With this form of communication, the sender and receiver do not engage in a send/receive simultaneously, which would require some kind of transactional infrastructure across the participants. Where the participants in a collaboration are geographically distributed a transactional infrastructure is complex and difficult to operate, so an asynchronous messaging infrastructure is more practical. For completeness of the theoretical treatment, we also consider synchronous collaborations, but not in as much detail.

The structure of the argument is not simple, as it involves a number of threads of discussion that are developed independently and then brought together. The proof extends over Sects. 6, 7, 8 and 9. Figure 6 provides a guide to the threads of the argument and how they combine to deliver the result.

6 Choreography realizability

In this section, we establish sufficient conditions for *realizability* of a choreography in both synchronous and asynchronous collaborations.

6.1 Preliminaries

We assume that we have a set of participants, $\mathfrak{P}$. We will use P, Q, R to refer to individual participants in $\mathfrak{P}$; and Pj, $j = 1, \ldots, n$, to index over all participants in $\mathfrak{P}$.

We also assume that we have a choreography, $\mathfrak{C}$, defined as a composition of independent machines (using Π to denote CSP $\|$ composition over a population of processes):

$$\mathfrak{C} = \prod_i \mathbf{C}^i$$

We will use **bold font** to indicate an independent machine (such as $\mathbf{C}$, $\mathbf{C}^i$ and $\mathbf{C}_P$), and normal font (such as C, C^i and C_P) to indicate a machine that may or may not be independent.

We further suppose that each machine, $\mathbf{C}^i$, in $\mathfrak{C}$ is projected to each participant $Pj \in \mathfrak{P}$ to give a projected machine $\mathbf{C}^i_{Pj}$. The projected contract for a participant, $Pj \in \mathfrak{P}$, is then:

$$\mathbf{C}_{Pj} = \prod_i \mathbf{C}^i_{Pj} \tag{6.1}$$

We assume that the projection of the independent choreography machine $\mathbf{C}^i$ to a participant P results in a contract machine for P that is also independent. This assumption is justified later, in Sect. 7.4.

This projection, along with the standard stipulation that a choreography fully constrains all receive actions in Pj, forms a protocol contract for each $Pj \in \mathfrak{P}$.

6.2 Definition of realizability

We define *choreography realizability* of a choreographed collaboration to mean:

- (C1) At any point of the collaboration it is possible to determine the corresponding state of the choreography.
- (C2) At any point of the collaboration, those sends allowed in the corresponding state of the choreography may happen in the collaboration, and only those sends.
- (C3) If a message is sent it is guaranteed to be received.

Before considering realizability, we define the notions of *faithful projection* and *trace matching*. We then show that, if these two properties hold, the choreography is realizable.

6.3 Faithful projection

We define the concept of *faithful projection*. Consider:

- A choreography machine: $\mathbf{C}^i$ in $\mathfrak{C}$.
- The machine $\mathbf{C}^i_P$ obtained by projecting $\mathbf{C}^i$ to $P \in \mathfrak{P}$.

Suppose $\mathtt{t} \in traces(\mathbf{C}^i)$, we define the *restriction* $\mathtt{t}\lceil_P$, of $\mathtt{t}$ to $\mathbf{C}^i_P$ as the sub-sequence comprising entries in $\mathtt{t}$ that involve P as either sender or receiver. An entry such as `P>Q:m++` in $\mathtt{t}$ having P as sender becomes `!>Q:m++` in $\mathtt{t}\lceil_P$, and an entry such as `Q>P:n++` in $\mathtt{t}$ having P as receiver becomes `?<Q:n++` in $\mathtt{t}\lceil_P$.

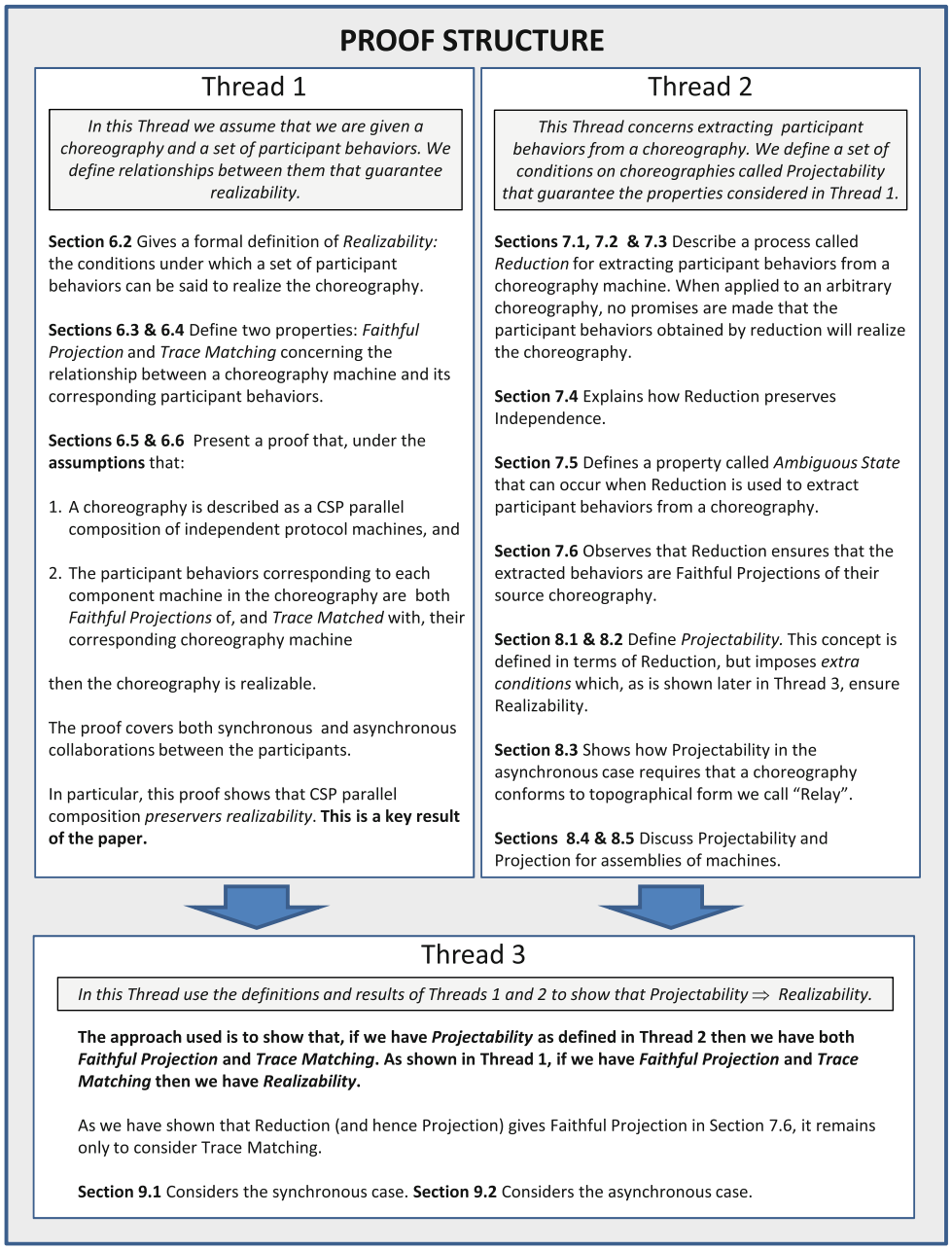

Fig. 6 Realizability proof structure

The projection is faithful iff:

$$\text{(F1)}\ \ \mathtt{t} \in traces(\mathbf{C}^i) \Rightarrow \mathtt{t}\lceil_P \in traces(\mathbf{C}^i_P).$$

Informally, faithfulness means that a projected machine can support any behavior (trace) that its source choreography machine allows. Faithful projection is a property of the way in which a choreography is projected to the participants, and we show that this property holds when we define the projection mechanism in Sect. 7.6.

6.4 Trace matching

We define the concept of *trace matching* in a choreographed collaboration. Consider:

- A choreography machine: $\mathbf{C}^i$ in $\mathfrak{C}$.
- The machines $\mathbf{C}^i_{Pj}$ obtained by projecting $\mathbf{C}^i$ to all $Pj \in \mathfrak{P}$.

Let $\mathcal{E}$ denote any (perhaps incomplete) execution of the collaboration between the projected machines. Suppose that the

collaboration executes to a global clock, and on each tick of the clock:

- In a synchronous collaboration, a single atomic send/receive takes place.
- In an asynchronous collaboration, either a single send or a single receive takes place.

This device gives a global sequence, $\mathtt{s}_{\mathcal{E}}$, of the sends executed in $\mathcal{E}$. We denote the restriction of this global sequence of sends to the alphabet of a given choreography machine, $\mathbf{C}^i$, by $\mathtt{s}^i_{\mathcal{E}}$. So a send of `m++` by P to Q of the global sequence is included as an entry `P>Q:m++` in $\mathtt{s}^i_{\mathcal{E}}$ iff $(P{>}Q{:}m) \in \alpha(\mathbf{C}^i)$.

The machines $\mathbf{C}^i_{Pj}$ are *trace matched* iff, $\forall$ collaborations $\mathcal{E}$:

(M1) $\mathtt{s}^i_{\mathcal{E}} \in traces(\mathbf{C}^i)$. $\mathtt{s}^i_{\mathcal{E}}$ is called the *implied trace* of $\mathbf{C}^i$ in $\mathcal{E}$.
(M2) $\forall j$ the trace of $\mathbf{C}^i_{Pj}$ in $\mathcal{E}$ is a prefix of $\mathtt{s}^i_{\mathcal{E}}\restriction_{Pj}$.

Informally, trace matching means that the sequencing of sends in a collaboration uniquely defines the traces (sends and receives) of the projected machines. Trace matching is a property of both a choreography and the way it is projected to the participants, and we will be considering the conditions under which this property is guaranteed later, in Sect. 8.

The ability to identify the state of the choreography machines on the basis of the observable sequence of sends meets the first condition, (C1), for realizability. We now go on to show that, if we have faithful projection and trace matching, we also meet conditions (C2) and (C3). We show this for first for synchronous and then asynchronous collaboration. In these arguments we take faithful projection and trace matching as given.

6.5 Synchronous realizability

We show that, in a **synchronous** collaboration, the combination of faithful projection and trace matching guarantees realizability of the choreography. We assume that the choreography is projected as described in (6.1) and that $\forall i, j$ $\mathbf{C}^i_{Pj}$ are faithfully projected and $\forall i$ $\mathbf{C}^i_{Pj}$ are trace matched. We may assume that the participants' designs contain their respective contract machines as, if not, we may add the contract machines without altering the behavior of the participant.

As this is a synchronous collaboration, every exchange in any trace of a choreography machine $\mathbf{C}^i$ must be fully represented (both send and receive entries present) in the traces of the projections. This means that no entries can be removed by the prefixing in (M2). So, if the sequence of exchanges so far in a synchronous collaboration, $\mathcal{E}$, is $\mathtt{s}$, the trace of $\mathbf{C}^i_{Pj}$ is **exactly** $\mathtt{s}^i_{\mathcal{E}}\restriction_{Pj}$.

Sending (C2). We assume the collaboration is successful up to some point with a sequence, $\mathtt{s}$, of synchronous exchanges. By (M1), $\forall i$, $\mathtt{s}$ implies a trace $\mathtt{s}^i$ of the choreography machine $\mathbf{C}^i$. Suppose that, $\forall j$ for which $(P{>}Q{:}m) \in \alpha(\mathbf{C}^j)$, $P{>}Q{:}m$ is possible for $\mathbf{C}^j$ after executing $\mathtt{s}^j$. Consider one of these machines, $\mathbf{C}^k$. We know that $\mathtt{s}^k{}^\frown\mathtt{P{>}Q{:}m{+}{+}} \in traces(\mathbf{C}^k)$.

By (F1) we have:

$$(\mathtt{s}^k{}^\frown\mathtt{P{>}Q{:}m{+}{+}})\restriction_P = (\mathtt{s}^k\restriction_P{}^\frown\mathtt{!{>}Q{:}m{+}{+}}) \in traces(\mathbf{C}^k_P)$$

By (M2) $\mathtt{s}^k\restriction_P$ is the trace so far of $\mathbf{C}^k_P$, so `!>Q:m++` is possible for $\mathbf{C}^k_P$. This applies to all machines in P required to participate in the send of `m++` under CSP $\|$ composition, so the send is possible for P as a whole.

In addition, (M2) means that a send that is not allowed by the choreography is not allowed by any participant, so we have (C2).

Receiving (C3). An identical argument shows that `?<P:m++` is possible for Q as a whole.

This means that these two machines can engage in a synchronous exchange on `P>Q:m++`. As the receipt by Q is fully constrained by the choreography contract, no other (non contract) machine in Q can refuse (block) it. This means that the choreography $\mathfrak{C}$ is realizable.

6.6 Asynchronous realizability

We now show that realization of the choreography is also guaranteed in an **asynchronous** collaboration between faithfully projected trace matched machines. As in the synchronous case, we assume that the choreography is projected as described in (6.1) and that $\forall i, j$ $\mathbf{C}^i_{Pj}$ are faithfully projected and $\forall i$ $\mathbf{C}^i_{Pj}$ are trace matched. Again, we may assume that the participants' designs contain their respective contract machines as, if not, we may add the contract machines without altering the behavior of the participant.

The method of argument is similar to the synchronous case. Note that, in the asynchronous case, while the traces of $\mathbf{C}^i_{Pj}$ may be matched for a given i, the traces for two different values of i may be interleaved differently at different participants so that we do **not** have trace matching on the contracts considered as a whole.

Sending (C2). We suppose an asynchronous execution $\mathcal{E}$ that has executed a sequence, $\mathtt{s}$, of sends. By (M1), this implies a trace $\mathtt{s}^i$ of each choreography machine $\mathbf{C}^i$. Consider an exchange $P{>}Q{:}m$ involving a message `m++` and suppose that $\forall j$ for which $(P{>}Q{:}m) \in \alpha(\mathbf{C}^j)$, $P{>}Q{:}m$ is

possible for $\mathbf{C}^j$ after executing $\mathtt{s}^j$. Consider one of these machines, $\mathbf{C}^k$. By (F1), $(\mathtt{s}^k{}^\frown \mathtt{!>Q:m++})\restriction_P \in traces(\mathbf{C}^k_P)$. This means that if the send is possible in the choreography, it is also possible in the collaboration.

Note that, unlike the synchronous case, (M2) only guarantees that current trace of $\mathbf{C}^k_P$ is a **prefix** of $(\mathtt{s}^k{}^\frown \mathtt{!>Q:m++})\restriction_P$ and there may be outstanding messages for $\mathbf{C}^k_P$ to receive before $\mathtt{!>Q:m++}$ becomes available to execute. So, unlike the synchronous case, the send may not be available to P immediately. This is inherent to asynchronous collaboration.

In addition, (M2) means that a send that is not allowed by the choreography is not allowed by any participant, so we have (C2).

Receiving (C3). Now we suppose that there is at least one message that fails: in that it cannot be successfully received by the participant to which it has been sent and so remains permanently "in flight". In the global sequence of sends in $\mathcal{E}$ we assume, without loss of generality, that the first failed message is $\mathtt{m++}_{fail}$ of type m sent by P to Q. We argue to show that this failure is impossible.

The two machines that govern the sending (by P) and receiving (by Q) of $\mathtt{m++}_{fail}$ are as below:

$$\mathbf{C}_{Pm} = \prod_{(!>Q:m)\,\in\,\alpha(\mathbf{C}^i_P)} \mathbf{C}^i_P \quad \text{and} \quad \mathbf{C}_{Qm} = \prod_{(?<P:m)\,\in\,\alpha(\mathbf{C}^i_Q)} \mathbf{C}^i_Q$$

The proof is based on the observation that at a point of sending $\mathtt{m++}_{fail}$, the component machines in $\mathbf{C}_{Pm}$ must all be synchronized under their CSP composition at states able to participate in the send; and similarly, at a point of receiving $\mathtt{m++}_{fail}$, the component machines in $\mathbf{C}_{Qm}$ must all be synchronized at states that are able to participate in the receive.

By (M1), the global sequence of sends up to and including the send of $\mathtt{m++}_{fail}$ gives an implied trace for each component machine, $\mathbf{C}^i$, of the choreography. We denote the implied trace of $\mathbf{C}^i$ by $\mathtt{t}^i_{fail}$. In addition, by (M2), the sequence of sends determines the trace (of both sends and receives) of each machine projected from $\mathbf{C}^i$ for all message exchanges in $\mathtt{t}^i_{fail}$ up to and including that for $\mathtt{m++}_{fail}$.

We consider a machine $\mathbf{C}^k_P$ in $\mathbf{C}_{Pm}$ and its matching machine $\mathbf{C}^k_Q$ in $\mathbf{C}_{Qm}$. As $\mathbf{C}^k_P$ has sent $\mathtt{m++}_{fail}$, its trace must contain $\mathtt{!>Q:m++}_{fail}$. So the implied trace, $\mathtt{t}^k_{fail}$, of $\mathbf{C}^k$ must contain $\mathtt{P>Q:m++}_{fail}$ and the restricted trace $\mathtt{t}^k_{fail}\restriction_Q$ must contain $\mathtt{?<P:m++}_{fail}$.

By (M2), the trace of $\mathbf{C}^k_Q$ is a prefix of $\mathtt{t}^k_{fail}\restriction_Q$. As all messages sent before $\mathtt{m++}_{fail}$ succeed (do not remain permanently "in flight"), all entries in the trace of $\mathbf{C}^k_Q$ before $\mathtt{?<P:m++}_{fail}$ will also eventually succeed, otherwise there is an earlier failure than $\mathtt{m++}_{fail}$. So $\mathbf{C}^k_Q$ must advance to the point where it can only remain trace matched by accepting $\mathtt{?<P:m++}_{fail}$ as the next entry in its trace.

This applies to all machines in $\mathbf{C}_{Qm}$, so all of these machines must reach a state where they can do nothing except receive $\mathtt{m++}_{fail}$. Moreover, by assumption, all messages sent by P to Q before $\mathtt{m++}_{fail}$ succeed so will all eventually clear from the queue between P and Q; so $\mathtt{m++}_{fail}$ must reach the front of the queue. Finally, as the receipt of $\mathtt{m++}_{fail}$ is fully constrained by $\mathbf{C}_{Qm}$, no other (non contract) machine in Q can refuse it. It must therefore be accepted. This means that the choreography $\mathfrak{C}$ is realizable.

7 Preservation and reduction

We now move to a discussion of how a choreography is projected into participant contracts. The aim is to show that, under defined conditions, the projection is faithful and ensures trace matching. With these shown to be the case, the results above give us realizability. Before defining how a choreography is projected to participants, we define:

- Preservation of Attributes and Derivation,
- Reducibility and Reduction of a Stored State Machine,
- Reducibility and Reduction of a Derived State Machine.

We use the device of a *filter*, $\mathcal{F}_{\mathfrak{S}}$, defined in terms of a set of participants, $\mathfrak{S} \subseteq \mathfrak{P}$. The effect of the filter is to remove any transition that does not involve **every** element of $\mathfrak{S}$ as either sender or receiver. So:

- $\mathcal{F}_\emptyset$ has no effect.
- $\mathcal{F}_{\{P\}}$ removes any transition that does not involve P.
- $\mathcal{F}_{\{P,Q\}}$ removes any transition that does not involve both P and Q.
- $\mathcal{F}_{\mathfrak{S}}$ is not defined if $\mathfrak{S}$ contains more than two elements.

The idea, of course, is that filtering a choreography using $\mathcal{F}_{\{P\}}$ is the basis for creating the projected contract for P. In the context of asynchronous collaboration, in Sect. 8.2, we will also need to consider filtering to pairs of participants.

7.1 Preservation of attributes and derivation

An attribute a is *preserved* under $\mathcal{F}_{\mathfrak{S}}$ iff no transition that carries (i.e., has a bubble attached containing) an update to a is removed by $\mathcal{F}_{\mathfrak{S}}$. A derivation function (that calculates a derived attribute, a derived state, or a field of an output message) is preserved under $\mathcal{F}_{\mathfrak{S}}$ iff every attribute used in the derivation is preserved under $\mathcal{F}_{\mathfrak{S}}$. A derived attribute is preserved under $\mathcal{F}_{\mathfrak{S}}$ iff its derivation function is preserved.

Note that preservation is defined in terms of a choreography **as a whole**. This is because, for instance, the derivation function for a derived state in machine C_1 might depend on

attributes in C_2 and C_3. Provided the attributes in C_2 and C_3 are preserved, the derivation function is too.

7.2 Reduction of a stored state machine

The stored state machine C is *reducible* under $\mathcal{F}_{\mathfrak{S}}$ iff:

- For every pair σ_a and σ_b (with possibly $\sigma_a = \sigma_b$) of states in C joined by a connected path that is completely removed by $\mathcal{F}_{\mathfrak{S}}$, **all** connected paths between σ_a and σ_b are completely removed. For this purpose the direction of the transitions is ignored: only the connectivity is considered.
- For every message send by C not removed by $\mathcal{F}_{\mathfrak{S}}$, all field derivations present in C are preserved by $\mathcal{F}_{\mathfrak{S}}$.

If C is reducible under $\mathcal{F}_{\mathfrak{S}}$, we have a relation on the states of C whereby two states are related iff the filter removes all the connections between them. This is an equivalence relation, as follows:

- We define a state as relating to itself, whether or not circular connections involving the state are removed, so the relation is reflexive.
- As the removal of a path is irrespective of the direction of the transitions that form the path, the relation is symmetric.
- If σ_a is related to σ_b and σ_b is related to σ_c, then there is a connected path from σ_a to σ_c that is completely removed, so σ_a is related to σ_c. This makes the relation transitive.

The *reduction* of C under $\mathcal{F}_{\mathfrak{S}}$, $C_{\mathfrak{S}}$, is a new machine created from C as follows:

- The states of $C_{\mathfrak{S}}$ are the equivalence classes of the relation defined by the filter.
- The transitions of $C_{\mathfrak{S}}$ are the transitions of C that are not removed by the filter. The start (end) state of a transition in the new machine is the equivalence class to which the start (end) state in C belongs under the relation.
- The attributes of $C_{\mathfrak{S}}$ (stored and derived) are all those in C that are preserved under $\mathcal{F}_{\mathfrak{S}}$.

If all transitions in C are removed by $\mathcal{F}_{\mathfrak{S}}$, the reduction is null (has no alphabet).

7.3 Reduction of a derived state machine

The derived state machine C is *reducible* under $\mathcal{F}_{\mathfrak{S}}$ iff:

- Either all transitions in C are removed by $\mathcal{F}_{\mathfrak{S}}$ or the derivation function of C is preserved by $\mathcal{F}_{\mathfrak{S}}$.
- For every message send by C not removed by $\mathcal{F}_{\mathfrak{S}}$, all field derivations present in C are preserved by $\mathcal{F}_{\mathfrak{S}}$.

The reduction of C under $\mathcal{F}_{\mathfrak{S}}$, $C_{\mathfrak{S}}$, is a new machine created from C as follows:

- The state derivation function and states of $C_{\mathfrak{S}}$ are the same as those of C.
- The transitions of $C_{\mathfrak{S}}$ are the transitions of C that are not removed by the filter. The transitions in the new machine have the same start state as in C.
- The attributes of $C_{\mathfrak{S}}$ (stored and derived) are all those in C that are preserved under $\mathcal{F}_{\mathfrak{S}}$.

If all transitions in C are removed by $\mathcal{F}_{\mathfrak{S}}$, the reduction is null (has no alphabet).

7.4 Preservation of independence

The reduction of an independent machine is also independent. This is because:

- The attributes of a reduced machine are those that are preserved by the reduction.
- We require that state and field derivations are preserved for reducibility.

This justifies the assumption made earlier, in Sect. 6.1.

7.5 Ambiguous states

Suppose **C** is an independent stored state machine. A state in the reduced machine, $\mathbf{C}_{\mathfrak{S}}$, corresponding to an equivalence class of the reduction filter $\mathcal{F}_{\mathfrak{S}}$ that contains more than one element is called an *ambiguous state*. An ambiguous state in a reduced machine is one which does not uniquely identify the state of the choreography machine from which it was derived.

States of a reduction of a derived state machine are never ambiguous. This is because reducibility of a derived state machine requires that the state derivation function is preserved in reduction to both the sender and receiver of any exchange in the machine (to conform to the semantics described in Sect. 5.3), so the reduction must have certain knowledge of the state value at all times in the collaboration.

7.6 Reduction faithfulness

Section 6.3 defines a notion of faithfulness in projection from a choreography, requiring that the restriction of a trace of the choreography to a participant P should always yield a trace of the projection of the choreography to P. It is clear that

the reduction rules above will always yield a reduction that is faithful to the choreography machine that is its source.

7.7 Reduction used for projection

Reduction of the machines of a choreography to the participants of the collaboration is the basis for *projection*, whereby protocol contracts for the participants are constructed. So the projection, $\mathbf{C}_P$, of a machine $\mathbf{C}$ is $\mathbf{C}_{\{P\}}$. Informally, a choreography being *reducible* to P (i.e., reducible under $\mathcal{F}_{\{P\}}$) means that its reduction to P gives P a coherent understanding of the state of the collaboration at all times, albeit potentially ambiguous in some states given that it does not know about any of the message exchanges in which it is not involved as either sender or receiver.

Clearly, projection requires that the choreography is reducible to each of the participants. However, reducibility of the machines of a choreography is not sufficient to ensure realizability, and we therefore make a distinction between *reducibility* and *projectability*, the latter including the extra conditions required to ensure realizability.

8 Projectability

The further conditions, beyond reducibility, required to ensure realizability depend on whether we are aiming for synchronous or asynchronous implementation.

We assume that the choreography, $\mathfrak{C}$, between a set of participants, $\mathfrak{P}$, is defined as a composition of independent machines:

$$\mathfrak{C} = \prod_i \mathbf{C}^i \tag{8.1}$$

Assuming that, $\forall i$, $\mathbf{C}^i$ is reducible to every participant in $\mathfrak{P}$ and that the contract, $\mathfrak{C}_P$, for a participant, $P \in \mathfrak{P}$, is given by:

$$\mathfrak{C}_P = \prod_i \mathbf{C}^i_P \tag{8.2}$$

where $\mathbf{C}^i_P$ is the reduction of $\mathbf{C}^i$ under $\mathcal{F}_{\{P\}}$.

We define two forms of projectability: synchronous and asynchronous. A choreography that has synchronous projectability guarantees that the choreography will be realized under synchronous communication between the participants, and asynchronous projectability guarantees that the choreography will be realized under asynchronous communication. Synchronous projectability is less exacting than asynchronous, and the difference between the two can be seen in Fig. 7, which needs to be read in conjunction with the definitions below. As our main interest is in the asynchronous case, we will consider in some detail how asynchronous projectability of a choreography is established.

8.1 Synchronous projectability

A choreography, $\mathfrak{C}$, between a set of participants, $\mathfrak{P}$, is *synchronous projectable* iff:

(S1) $\forall i$, $\mathbf{C}^i$ is reducible to every participant in $\mathfrak{P}$ and the reductions are deterministic.
(S2) No transition for a send by any participant starts from an ambiguous state.

The first condition is required to ensure that we can form the projected contracts, as specified in (8.2). We require the reductions to be deterministic to qualify as protocol contracts. As we shall see in Sect. 9.1, these two conditions ensure that we have trace matching under synchronous collaboration, and therefore that the choreography is realizable.

8.2 Asynchronous projectability

A choreography, $\mathfrak{C}$, between a set of participants, $\mathfrak{P}$, is *asynchronous projectable* iff:

(A1) $\forall i$, $\mathbf{C}^i$ is reducible to **every pair of participants** in $\mathfrak{P}$ and the reductions are deterministic.
(A2) No transition for a send by any participant starts from an ambiguous state.
(A3) In a given state of a choreography machine, $\mathbf{C}^i$, only one participant may send.

As we shall see in Sect. 9.2, these three conditions ensure that we have trace matching under asynchronous collaboration, and therefore that the choreography is realizable. The reason for requiring reducibility to **every pair** in (A1) will become clear in the proof of trace matching.

First, we show that (A1) ensures that we also have reducibility to every participant and that such reductions are also deterministic, as is required if we are to use the reductions by participant as the basis for projection to contracts according to (8.2). Suppose that a machine C can be reduced to each pair but is nevertheless not reducible to P. We consider two cases:

– C is a stored state machine. Failure to reduce to P means that there are two paths between some pair of states, one of which is removed by $\mathcal{F}_{\{P\}}$ and one which is not. Suppose that an exchange in the path that is not removed involves P and Q. In the reduction to $\{P, Q\}$, this exchange is also preserved, so again the first path is removed but the second is not. This means that reduction to $\{P, Q\}$ is not possible either.

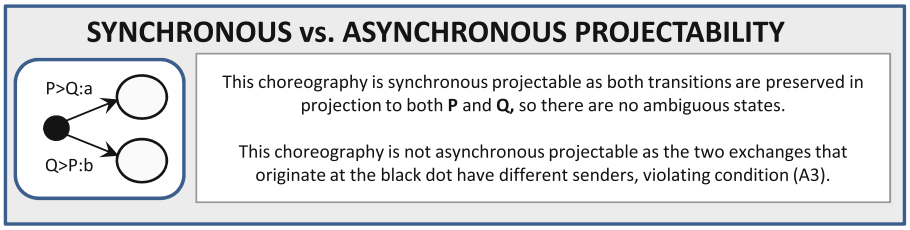

Fig. 7 Synchronous versus asynchronous projectability

– C is a derived state machine. Failure to reduce to P means that the state derivation function is not preserved by $\mathcal{F}_{\{P\}}$ and at least one exchange is not removed. If the derivation function is lost in reduction to P, it will also be lost in reduction to any pair including P. Suppose that the exchange not removed involves P and Q, this exchange will also be kept in reduction to $\{P, Q\}$. These means that reduction to $\{P, Q\}$ is not possible either.

Finally suppose that, in either the stored or derived state case, the reduction to P is non-deterministic with two identical transitions involving P and Q starting from the same state. The same non-determinism will be manifested in the reduction to $\{P, Q\}$.

So if a choreography is reducible to every pair and gives deterministic reductions, it is also reducible to every participant giving deterministic reductions.

Note that, although we require reducibility to every pair for asynchronous projectability, participant contracts are still based on reduction to individual participants. This is because, although reducibility to pairs is a more exacting requirement (as it is sufficient but not necessary for reduction to individuals), reduction to pairs loses more information than reduction to individuals. Consider a choreography in which P sends to Q which sends to R. The sequencing of the exchanges is preserved in reduction to Q but lost in all reductions to pairs.

8.3 Relay choreography

Because the asynchronous case is the more important in practice, we will now examine asynchronous projectability in some detail. Consider condition (A3). Suppose that σ is a state some machine $\mathbf{C}^i$ from which P is the only sender, and there is at least one exchange sent by P starting at σ. Suppose that σ is also the end state of a transition. If the exchange of the transition ending at σ does not involve P as either sender or receiver it is removed in reduction to P. This makes σ ambiguous, violating (A2). This reasoning shows that each machine, $\mathbf{C}^i$, in the choreography must be a *relay machine* defined as one with the following properties:

(R1) In any state that the machine can adopt, only one participant can send.

(R2) If two adjacent states (one reachable from the other by a single transition) have different senders, so that P is the sender in σ_1 and Q is the sender in σ_2, then all transitions directly from σ_1 to σ_2 must have P as sender and Q as receiver.

In the context of property (R2), note that:

– If there are two (or more) direct transitions from σ_1 to σ_2 they must involve different message types to ensure determinism.
– There may be direct transitions in both directions, and a direct transition from σ_2 to σ_1 must have Q as sender and P as receiver.

The term "relay" is by analogy with a relay race, in which a baton is passed from one runner to the next. Property (R1) says that only the participant "holding the baton" may send, and property (R2) says the participant must "pass the baton" to the next sender.[8] While in possession of the baton, a participant may pass through any number of states as sender and send any number of messages to different participants, before passing the baton to the next sender. Although all the transitions from a given state must all have the same sender, different transitions from the same state may have different receivers. Figure 8 shows an example of a relay trace choreography between three participants. Because machine equivalence is defined by trace equivalence, the relay property of a machine can be determined from its traces. Every trace must have the following property: if the sender of an entry in the trace is different from the sender in the previous entry, the sender of the second entry must be the receiver of the first.

A choreography is a *relay choreography* if it is a CSP $\|$ composition of independent relay machines. Note that, in general, a relay choreography allows a given participant at a given point in a collaboration both to send messages to different participants, and to receive messages. For example, suppose the choreography is a composition of three relay

[8] This analogy is not exact, as there is no representation of the baton in the choreography. A send event that "passes the baton" is syntactically and semantically identical to one that does not.

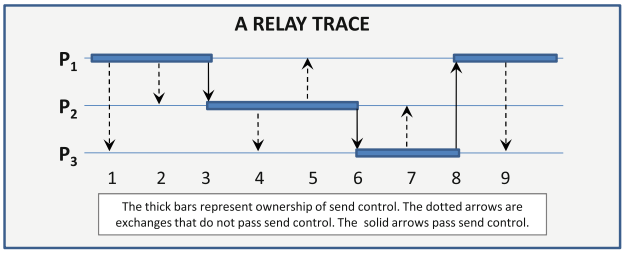

Fig. 8 Relay trace

machines, $\mathbf{CR}_1 \parallel \mathbf{CR}_2 \parallel \mathbf{CR}_3$ it is possible that, at some point in a collaboration:

- $\mathbf{CR}_1$ allows P to send; and
- $\mathbf{CR}_2$ allows Q to send with P as a receiver; and
- $\mathbf{CR}_3$ allows R to send with P as a receiver.

Each relay machine in the composition has its own baton, and you can have as many relay machines composed in the choreography as you like.

8.4 Relay analysis of assemblies

As demonstrated by the examples in Fig. 9, the rely property is not preserved by CSP composition: it can both created and destroyed. So it may be that a choreography is articulated as a set of "base" machines which, although not all are independent and relay individually, can be refactored so that it is expressed in the form given in (8.1), with some or all of the $\mathbf{C}^i$ being assemblies. If it is possible, such a refactoring establishes that the choreography as a whole is relay. To support such a process, we need mechanisms to check for the relay property in *assemblies*. We consider two cases: the first where the assembly only involves stored state machines and the second where the assembly involves derived state machines.

If an assembly consists only of stored state machines, we can use the construction described in Sect. 2.3 to create a single topological representation of the assembly which can then be checked to be relay. In the case that an assembly involves derived state machines, we cannot use the prescription in Sect. 2.3 directly as the technique described there only applies to stored state machines. However, for the purpose of analysis, it is possible to create a stored state approximation of a derived state machine called the *connected form* of the machine for topological analysis. This is done in three steps:

1. Form the state space as that defined by the machine's state function.
2. Add transitions between the states to create a topological surrogate for the state function.
3. Create transitions representing the constraints of the original machine, but in topologically connected form.

An example is shown in Fig. 10 in which a connected form representation, $A2^*$, is shown for the machine $A2$ from Fig. 3. Step 1 identifies the two states: {*in credit*, *overdrawn*}. In step 2, transitions (shown as dashed for graphical emphasis) are added as a surrogate for the state function. In step 3, the transition for *Close* is shown as both starting and ending at the state *in credit* as the action can only happen from this state and leaves the state unchanged. In some cases, step 3 may involve removing transitions added in step 2. Suppose that $A2$ were to constrain *Withdraw* only to happen in the state *in credit*, as it does *Close*, then the dashed transition for *Withdraw* starting from *overdrawn* added in step 2 would be removed in step 3.

Step 2 merits some elaboration. The transitions that need to be added in this step (those shown as dashed arrows) are identified as follows:

- The state function is examined to determine the set of attributes referenced in the calculation. In this case, it is just `A1.balance`.
- The machine(s) that own these attributes are examined to determine which transitions cause their values to change. In this case it is *Open*, *Deposit* and *Withdraw* in $A1$ as can be seen from the update bubbles attached to these transitions.
- Corresponding transitions are added to each state of the connected form machine being constructed, according to how they can alter the state. For instance, *Withdraw* decreases the value of `balance` so its effect on $A2^*$ is either to leave the state unchanged or to cause it to change from *in credit* to *overdrawn*.

Whereas the state function tells us **exactly** the effect of depositing or withdrawing funds on the state, the surrogate transitions show, and can only show, the **possibilities**. Generally, the transitions added in step 2 create a machine that is non-deterministic. While the left-hand, normal PM, machine in Fig. 10 is deterministic the right-hand figure is not. This is a consequence of collapsing the state space of the algebraic model from its original complete enumeration of the balance values and collapsing the value specific events $\{D1, D2, W1, W2\}$ into generic *Deposit* and *Withdraw* events. The connected form, however, is **only for analysis purposes**, and its non-determinism does not pose any barrier to the analysis. The connected form machine does not replace the derived state version in the definition of the choreography, nor is it used as the basis for projection to participant contracts.

We argue that the connected form can be used in place of the original derived state machine in the context of relay anal-

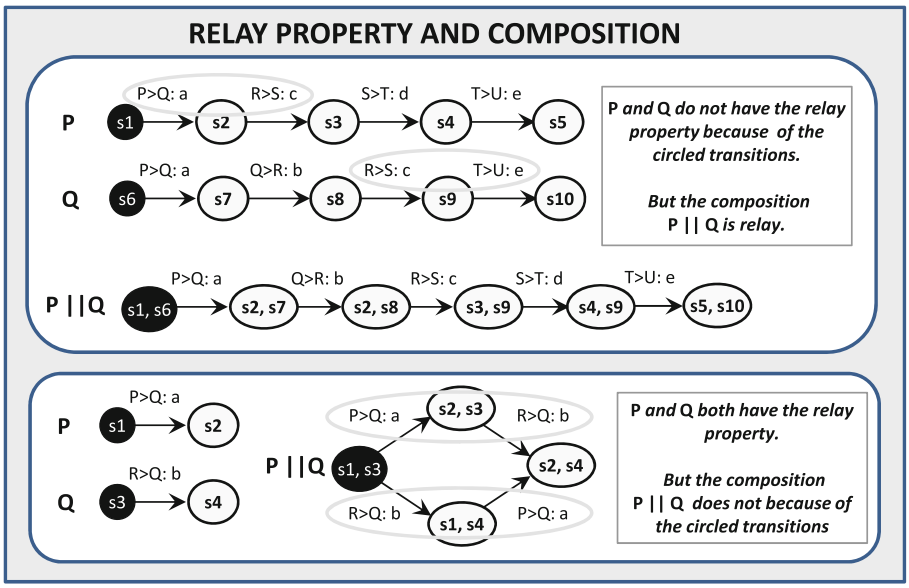

Fig. 9 Relay property and composition

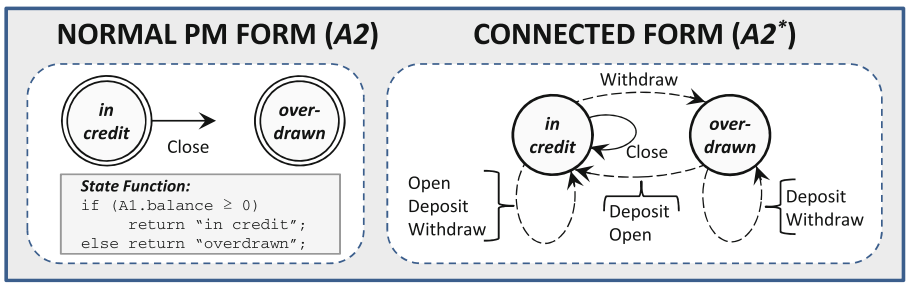

Fig. 10 Connected form of a derived state machine

ysis, as follows. As the connected form machine describes the transition possibilities of the machine it approximates, its traces are a superset of the traces that are possible for the original derived state machine. If the traces of the connected form display the relay property, the subset that constitutes the traces possible for the original derived state machine must also adhere to relay behavior. We may therefore use the connected form, combining it with other machines using the technique in Sect. 2.3 where required, to determine that an assembly involving derived state machines has the relay property. Figure 11 shows an example of relay analysis on a choreography involving a derived state machine. The choreography is defined as two machines, $C1$ and $C2$. Forming the connected form, $C2^*$, of $C2$ and composing this to form $C1 \parallel C2^*$ allows the relay property to be established.

8.5 Projection of assemblies

Once we have established that an assembly is relay, we need to be able to project the assembly to participants (to single participants and to pairs) both in order to complete the analysis of projectability by ensuring that projections are deterministic, and to create the participant contracts. Here, we have to face the fact that it is **not** true, in general, that the reduction of an assembly is equal to the assembly of the reduction of its components. As in the previous section, we consider first the case where the components are all stored state machines and then the case where the assembly involves derived state machines.

Where an assembly consists only of stored state machines, the machines in the assembly are combined using the construction described in Sect. 2.3 to create a topological representation of the assembly prior to reduction. Once expressed as a single machine, the recipe for reduction described in Sect. 7.2 can be used in the normal way.

Now suppose that the assembly involves a derived state machine. We cannot use the connected form as the basis for reduction as this only approximates the behavior of the original machine. Instead we show that, in this case, it is safe to perform reduction on the derived state machine independently of the other machines in its assembly. First we appeal to the definition of the semantics of a derived state machine

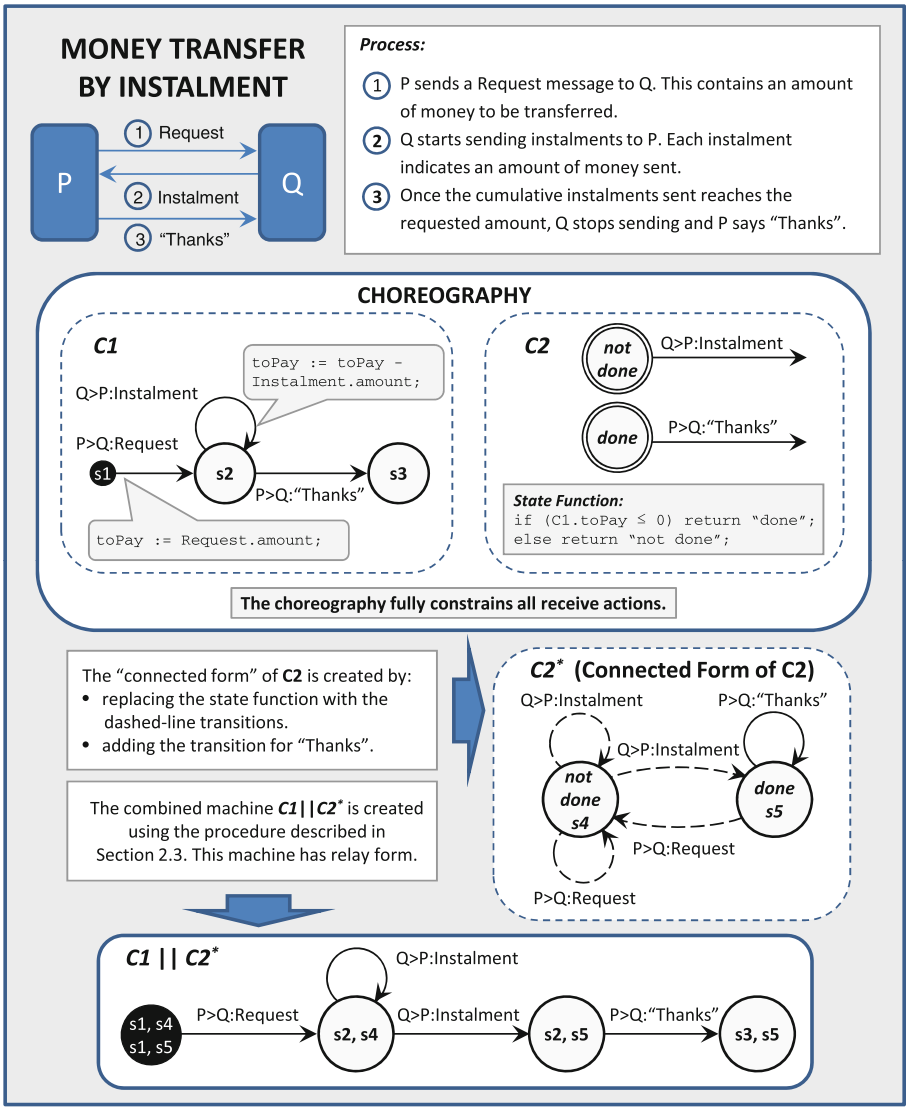

Fig. 11 Instalments example

in a choreography given in Sect. 5.3. Suppose that we have a transition between P and Q whose occurrence is constrained by a derived state machine, C_d. The semantics of a derived state machine in a choreography requires that both the sender, P, and the receiver, Q, are able to determine the state of C_d. This means that all transitions implicated in calculation of the state of C_d must be preserved in reduction to both P and Q, and therefore all such transitions must involve both P and Q.[9] Consequently, these transitions can involve no other participant. As the only transitions in C_d are those that are either constrained by C_d or implicated in calculation of the state of C_d, we have that the elements of the alphabet of C are all exchanges between P and Q. We can now invoke a result of Hoare's work on *concealment* in CSP. The relevant result concerns concealment of a set of actions c in the composition of P and Q, using "\" to denote concealment, and states that:

$$\text{If } \alpha(P) \cap \alpha(Q) \cap c = \emptyset \text{ then} \quad (P \parallel Q) \setminus c = (P \setminus c) \parallel (Q \setminus c) \tag{8.3}$$

Concealment in CSP can be equated with *removal by reduction* in our theory. The result (8.3) shows that the reduction of an assembly to a given participant is the same as the composition of the reductions of the components to that participant, provided that no action in the intersection of the alphabets of the components is removed by the reduction. As we have that all the transitions in C_d involve both P and Q, none is removed by reduction to either P or Q. It is therefore safe to reduce C_d to any of $\{P\}$, $\{Q\}$ or $\{P, Q\}$, using the recipe given in Sect. 7.3, separately from any other machine with which it is in assembly in the definition of the choreography. As C_d has a null reduction to any other participant, these

[9] This is equivalent to stating that the transitions added in step 2 of the procedure described in Sect. 8.4 involve only P and Q.

are the only reductions we need to consider. Any stored state machines in assembly with C_d must be combined with each other prior to reduction, as described above.

9 Proof of realizability

Section 6.6 showed that the combination of faithfulness and trace matching gives realizability. As we know projection is faithful (Sect. 7.6), we only need to show that the conditions for projectability given in Sect. 8 ensure trace matching. We address the synchronous case and the asynchronous case in turn.

9.1 Synchronous case

To prove realizability, we need to establish that (S1) and (S2) guarantee trace matching. We use induction on a machine $\mathbf{C}^i$ in the choreography. We suppose that at some point in the collaboration:

- There is a trace $\mathtt{t}^i$ of $\mathbf{C}^i$ so every participant, Pj, has $\mathtt{t}^i_{Pj} = \mathtt{t}^i \upharpoonright_{Pj}$.
- $\mathbf{C}^i$ is in state σ.
- Every $\mathbf{C}^i_{Pj}$ is in a state that corresponds under projection to σ.

Suppose that, at this stage of the collaboration, for some participant P, $\mathbf{C}^i_P$ allows P to send a message `m++` to another participant Q. By (S2) $\mathbf{C}^i_P$ is not in an ambiguous state, and by the induction assumption, must be in a state that corresponds 1-1 with σ. So the exchange `P>Q:m++` must be allowed at σ in $\mathbf{C}^i$. Because $\mathbf{C}^i_Q$ is also, by assumption, in a state (albeit possibly ambiguous) that corresponds to σ it must allow receipt of `m++`. Assume this exchange happens:

- `P>Q:m++` is added to $\mathtt{t}^i$; and
- `!>Q:m++` is added to the trace of $\mathbf{C}^i_P$; and
- `?<P:m++` to the trace of $\mathbf{C}^i_Q$.

As $\mathbf{C}^i_Q$ is deterministic, it advances to a new state still synchronized with $\mathbf{C}^i$. Induction begins when the choreography and all participants are at their initial states and all traces are empty.

This means that a projected machine is always synchronized with the state of choreography machine from which it was projected; and so:

- The sequence of sends in a collaboration match a trace of the choreography, giving (M1).
- The traces of a projected machine always matches the trace of corresponding choreography machine, giving (M2).

This establishes trace matching and realizability.

9.2 Asynchronous case

To prove realizability, we need to establish that (A1), (A2) and (A3) guarantee trace matching. We consider a single relay machine, $\mathbf{C}^i$, in the choreography. We assume a communication environment providing an unbounded FIFO queue in each direction between each pair of participants.

Figure 8 shows an execution scenario of a relay choreography machine. The discipline of baton passing means that the send operations are globally ordered along the bars, which represents the "relay baton". The global ordering of the sends defines the trace, $\mathtt{t}^i$, of the choreography machine. First, it is clear that the sequence of sends in which this choreography machine is involved define $\mathtt{t}^i$ and this gives (M1).

To establish (M2) we have to show that, in any scheduling of the collaboration and any participant P, the trace $\mathbf{C}^i_P$ is a prefix of $\mathtt{t}^i \upharpoonright_P$. To simplify the notation, we drop the ^{i} superscript and refer to the choreography machine $\mathbf{C}^i$ as $\mathbf{C}$ and correspondingly for the projected contract machine.

We establish trace matching by induction. Consider a choreography machine, $\mathbf{C}$, and its projection, $\mathbf{C}_P$, to P at some point in the collaboration:

- The trace of $\mathbf{C}$ so far is $\mathtt{t}$, and the current state of $\mathbf{C}$ is σ_t.
- Because of trace matching condition (M2) the trace, $\mathtt{t}_P$, of $\mathbf{C}_P$ so far is $\mathtt{t'} \upharpoonright_P$ where $\mathtt{t'}$ is a prefix of $\mathtt{t}$.
- The current state of $\mathbf{C}_P$ corresponds to the state $\sigma_{t'}$ that pertained in $\mathbf{C}$ after it had executed $\mathtt{t'}$.

We consider the next possible actions in which $\mathbf{C}_P$ can engage. Projection rule (A3) requires that only one participant is the current sender in $\mathbf{C}$. There are two possibilities:

- P is the current sender (holding the baton) in $\mathbf{C}$.
- P is not the current sender in $\mathbf{C}$.

If P is the current sender in $\mathbf{C}$ then, because of the baton passing discipline, $\mathtt{t'} = \mathtt{t}$. So $\sigma_{t'} = \sigma_t$. By projection rule (A2) a sender cannot be in an ambiguous state. The only next possible action by $\mathbf{C}_P$ is a send, and because it is not in an ambiguous state this must be send allowed by $\mathbf{C}$ at σ_t. The result of such a send is that corresponding entries are added to the traces of $\mathbf{C}$ and $\mathbf{C}_P$ and trace matching is preserved.

If P is not the current sender in $\mathbf{C}$ then $\mathbf{C}_P$ can only receive. Without loss of generality, we assume that there is at least one exchange in $\mathtt{t}$ beyond $\mathtt{t'}$ that sends a message to P and that the first such exchange is `Q>P:m++`. As all exchanges

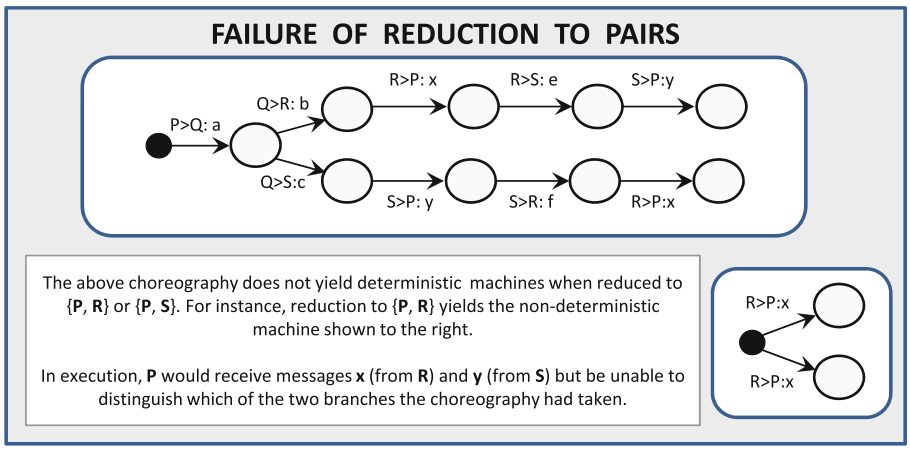

Fig. 12 Failure of reduction to pairs

involving P in $\mathtt{t}'$ have entries in $\mathtt{t}_P$, no message to P sent before m++ is still queued; and because the queues are FIFO, no message sent after m++ by Q to P can overtake it. So it must become available to P. As we have trace matching so far, P must be able to receive it.

We need to show that $\mathbf{C}_P$ is bound to stay matched to the trace t when it consumes a message. There are two possible causes for departure:

- $\mathbf{C}_P$ consumes the message m++ from Q but advances to a state that does not correspond to the state that **C** has after the Q>P:m++ exchange.
- Another message, n++ sent by R in an exchange that occurred in t after Q>P:m++ is accepted by $\mathbf{C}_P$ before m++, causing it to receive messages in an order that does not correspond to t and hence depart from match to t. An example of this situation is shown in Fig. 12. Note that such a message cannot come from Q, as it would then not be available to P before m++ because of the FIFO queueing discipline.

The first cause would require that there is a trace $\mathtt{t}_{false}$ of **C** such that $\mathtt{t}_{false}\restriction_P = \mathtt{t}_P$ + ?<Q:m++; and such that the state of **C** after Q>P:m++ in $\mathtt{t}_{false}$ is different from its state after Q>P:m++ in t. This requires that the exchange Q>P:m++ uses different transitions in the graph of **C** in the context of the two traces. However, in $\mathbf{C}_P$ these transitions are both available to P after it has executed $\mathtt{t}_P$ and this would make this reduction non-deterministic, which is not allowed by (A1).

The second cause would require that there is a trace $\mathtt{t}_{false}$ of **C** containing R>P:n++ followed, as the next send by Q to P, Q>P:m++. We consider the graph of **C** from the state after $\mathtt{t}'$ to the state where it executes R>P:n++. The path used by t has a send from Q to P and the path used by $\mathtt{t}_{false}$ does not. So these paths must lead to different states, otherwise **C** would not be reducible to $\{P, Q\}$, as required by projection rule (A1). If they are different states, then the two transitions used by R>P:n++ are different in the two paths. In the reduction of **C** to $\{P, R\}$ these would both be available in the state after t, and this would make this reduction non-deterministic, which is not allowed by (A1).

Induction begins when the choreography and all participants are at their initial states and all traces are empty. This establishes trace matching and realizability.

10 Example

As an example, we take the *Shared Book Buying Collaboration* described by Honda et al. [7]. In this example, there are three participants: a Seller (S) and two buyers ($B1$ and $B2$). Buyer1 can afford a portion of a book's price and the balance has to be paid by Buyer2, who also makes the buy or no buy decision. The choreography is shown at the top of Fig. 13 described as two composed machines. It is easy to see that the choreography is relay, so adheres to projection conditions (A2) and (A3). Moreover, as a given exchange only appears once in each choreography machine, any reduction must be deterministic so the choreography adheres to projection condition (A1). So asynchronous realizability is guaranteed.

Below the choreography in Fig. 13 is shown the extraction of the contract for the participant Buyer1. Similar extraction produces the three contracts shown in Fig. 14.

11 Related work

A number of other authors have looked at choreography definition and the rules required to ensure realizability. In this section, we discuss related work and provide a commentary on how this work relates to this paper. We focus on work on asynchronous choreographed collaborations.

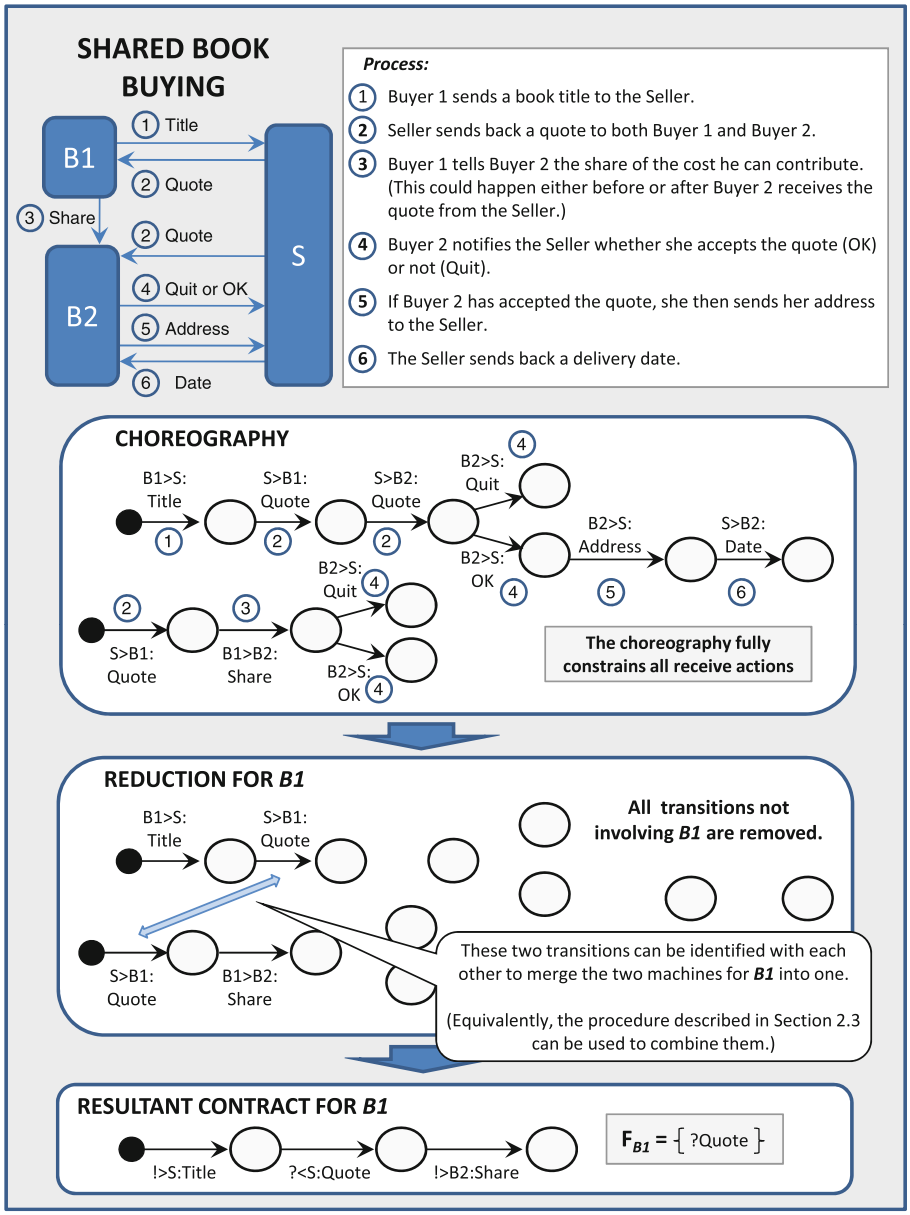

Fig. 13 Shared book buying: choreography and contract for Buyer1

Session typing. Work by Honda et al. [7] addresses the question of realizability of asynchronous multiparty collaborations using a behavioral typing discipline called *Session Typing*. The scheme defines *global types*, which represent choreography; and *local types*, which correspond to our participant contracts. The local types are abstracted from the global type by a projection calculus, similar to that we describe in Sect. 7. The conditions we give for asynchronous projectability are captured in their notion of *linearity*, which aims to enforce the discipline that *in two communications, sending actions and receiving actions should respectively be ordered temporally, so that no confusion arises*. Linearity is established by a process called *causality analysis* based on the dependencies between the exchanges of the global type. This analysis requires that there are subsequences in the choreography that observe a similar discipline to that we require in relay machines.

Where there is indeterminacy of ordering in the global type, sequencing is imposed using a device called a *channel*. In their solution to the *Shared Book Buying Collaboration*, which we discussed in Sect. 10, the channel device is used to address the indeterminacy in the ordering of receipt of the *Quote* and *Share* messages by $B2$ by imposing an ordering: *Quote* followed by *Share*. This is not required in the solution we present, and it seems that channels are used to remove the natural concurrency of the collaboration which in our approach is expressed explicitly using composition in the choreography. This means that the Session Typing approach,

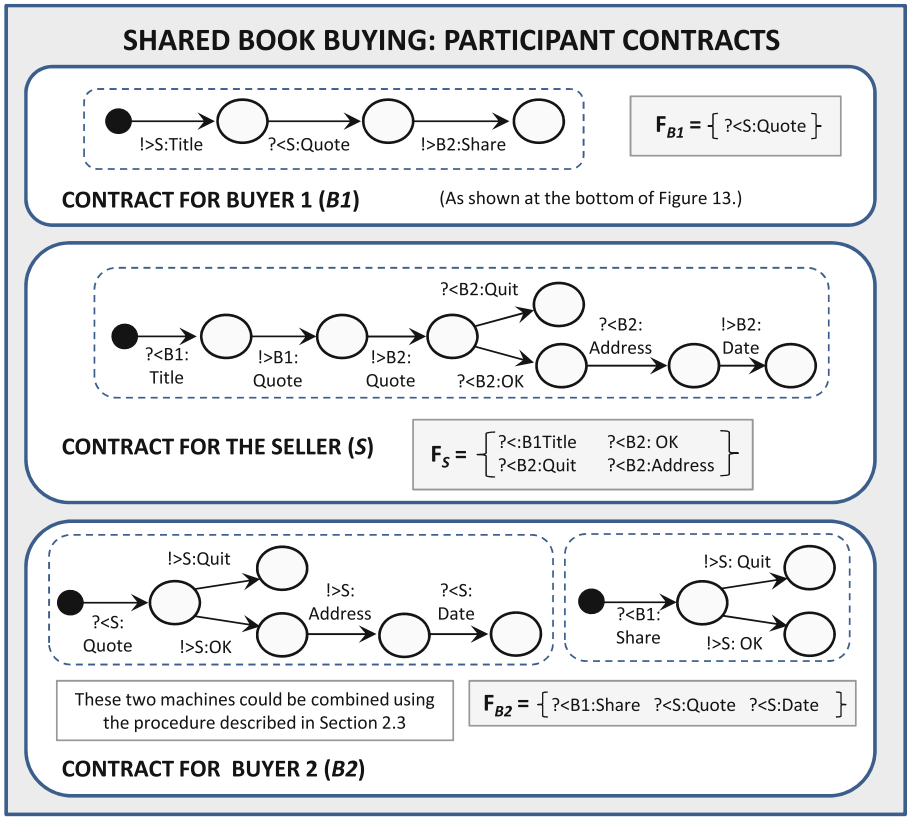

Fig. 14 Shared book buying: participant contracts

at least as described, is not able to describe collaborations that necessarily entail a race. An example is given in Fig. 15 which describes a simple competition in which two contestants, X and Y, are asked a question and a prize is awarded for correct answers. A choreography expressed as four composed protocol machines is shown in Fig. 15, and we note the following:

- The collaboration involves a race, because the sequencing of the receipt of answers from X and Y is unknown in advance and cannot be constrained by the choreography. This is handled using a composition of two machines in the choreography, shown one above the other at the bottom left of the figure, allowing the ordering of receipt of the answers from X and Y to be unspecified.
- The provision of the *Decision* by Q back to P and then by P on to X and Y is depicted by the two machines at the bottom right of the figure. Note that it is possible for P to send notification of the decision to X or Y before they have responded, as it is possible that either or both failed to respond quickly enough.
- The four machines in the choreography are independent, relay and deterministic when reduced to pairs. So the choreography is projectable and therefore realizable under our theory.

In attempting to create of a global type for this collaboration, we have a choice about how to express the sending of the question to X and Y and the subsequent receipt of their answers by Q. One possibility is to use two session types composed in parallel, following the same strategy that is used in Fig. 15. However, while the syntax of global types allows parallel composition, the projection of a global type that involves parallel composition is undefined if a given participant is involved in both the types being composed, either as the sender or the receiver of a message, so this approach is excluded. The other possibility is to serialize the two exchanges whereby X and Y send their answers to Q in a single global type. In this case, we need to address the question: Should these two exchanges use the same or different channels? Neither choice appears to be satisfactory. If we choose to use different channels then P is forced to receive the answers in a defined order, as the receives on the channels must be serialized in P's logic. This clearly violates the intention of the collaboration. If we choose to use the same channel then the global type fails to obey the condition for *linearity*[10] (the Session Typing equivalent of our projectability condition), and this means that realizability is not guaranteed. As some collaborations, such as e-auctions, are likely to involve such intentional race conditions the difficulty of modeling this using Session Types seems a weakness.

[10] This is because a chain of two receipts on the same channel from different senders, as we would have here, is considered "non-causal" and makes the global type non-linear. See Honda et al. [7]

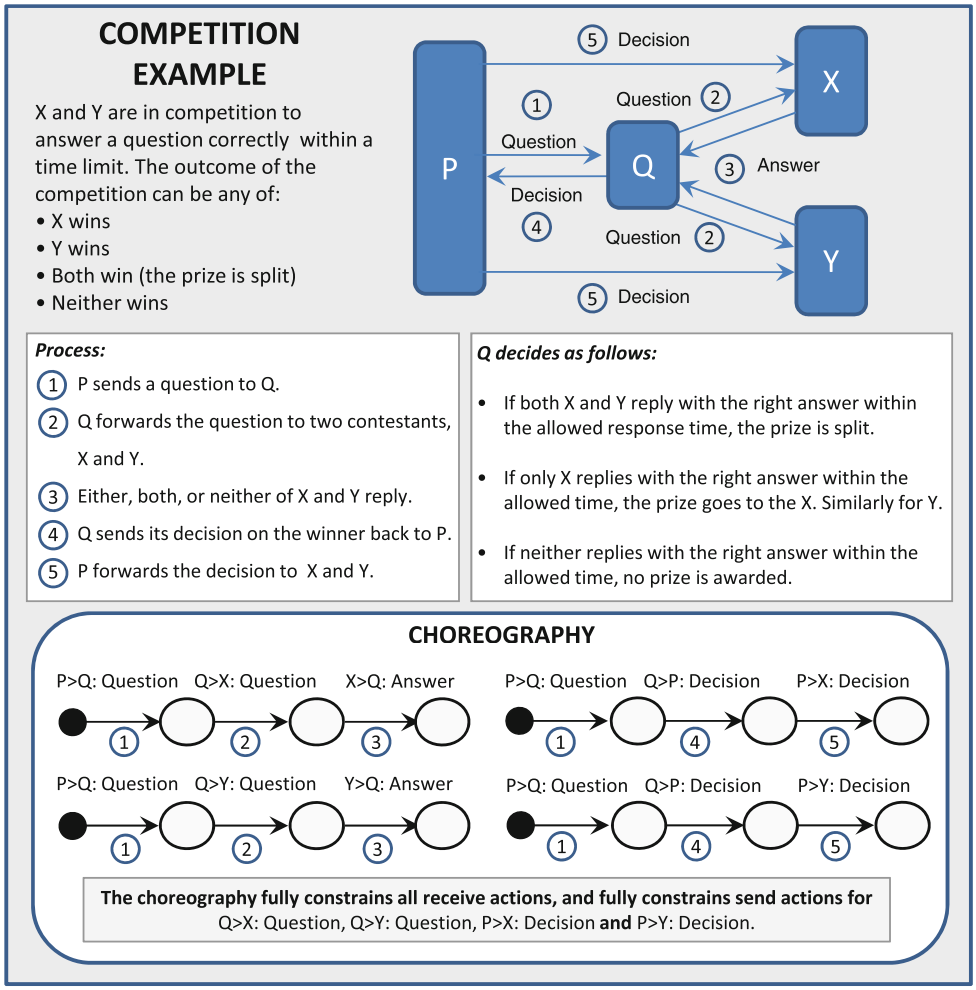

Fig. 15 Competition example

More recent work is underway, [1], to extend the Session Typing to include a treatment of data. This work aims to allow contractual assertions to be included in the definition of global types and thus bring a "Design by Contract" [12] capability to protocol construction. At present, this work is not aimed at allowing data and computation to be used for the definition of choreographical behavior constraints and so does not compare directly with our work.

IOC/POC. Work by Lanese et al. [9] define an approach similar to that used in Session Typing, but without the use of channels. The scheme defines *interaction-oriented choreography* (IOC), which represents choreography; and *process-oriented choreography* (POC), which corresponds to our participant contracts. A POC for each participant is abstracted from the IOC by a projection calculus. The authors give realizability conditions, which they call *connectedness conditions*, for both synchronous and asynchronous collaborations. These conditions correspond very closely to those we describe for projectability of a single stored state choreography machine.

The IOC/POC scheme allows parallel composition of processes in a choreography and, unlike Session Typing, allows the same participants and message types to appear in different composed processes. This means that the IOC/POC scheme has the ability to model concurrency involving races and could be used to express the choreography for the Competition Example in Fig. 15. However there is no concept of CSP style synchronization or of the use of data and derived states, so it is not possible to articulate a choreography as a set of separate partially synchronized descriptions. This means that the approach would not, as currently described, be able to describe a choreography that requires a combination of stored and derived state spaces such as the Instalments Example in Fig. 11 and, as published so far, the IOC/POC approach does not use data or computation at all.

Conversation protocols. In their work, Fu et al. [3] describe an approach for the realization of asynchronous collaborations

based on *Conversation Protocols*. A conversation protocol corresponds to a choreography and is projected to give peer (= participant) behavior. Both the choreography and the projected peer behaviors are modeled as Finite State Automata (FSA). The authors identify three conditions on a conversation protocol that must be met for realizability: *Lossless Join* which requires that the protocol should be complete when projected to individual peers, *Synchronous Compatible* which ensures that the protocol does not have illegal states, and *Autonomy* which implies that at any point in the execution each peer is determined on the choice to send, or to receive, or to terminate. Although stated in different terms, these correspond quite closely to our asynchronous projectability conditions. The Autonomy condition allows more than one peer (participant) to be in send mode at the same point in the choreography so is more relaxed than our projectability condition (A3) which limits sending to a single participant. This more relaxed condition is balanced by the stricter requirement of Lossless Join to ensure realizability. As we point out in Sect. 8.3, different machines in our compositional approach can have different senders, and this gives our approach equivalent expressive power to Conversation Protocols.

This formalism used for Conversation Protocols does not use parallel composition at all. This does limit expressive power. For instance, if:

$$(P > Q{:}a) \parallel (Q > P{:}b)$$

is expressed as a Conversation Protocol (i.e., with no composition) then it violates Autonomy as P and Q are both sending and receiving in the initial state of the choreography. However, both components are clearly asynchronously projectable, so the choreography is realizable under our theory. The same issue would arise in attempting to model the Customer and Supplier choreography in Fig. 5 as a Conversation Protocol as, for instance, there is a state where the *Customer* can send a *Request Cancel* and also receive an *Invoice*.

In more recent work, [4], the authors have added some capability to model data and computations. This is done by using *Guarded* Finite State Automata (GFSA) for both choreography and projected peer behaviors. The general idea is similar to that we describe, with data and computation being used to specify rules in the choreography. However, the scheme is significantly less ambitious than ours in that the guard of a transition only expresses the relationships between the message that is being sent and the last message of each class sent or received by the sender; so there is no notion of a choreography owning and maintaining attributes as we have. Updates are confined to changes to message fields, and it would not be possible to compute a guard condition on accumulated amounts[11] such as is required for the Instalments Example in Fig. 11. The authors describe an algorithm for checking realizability of a choreography described as a GFSA, and this bears some similarities to the technique described in Sect. 8.4 but a detailed comparison is beyond the scope of this paper.

BPMN2. The *Business Process Model and Notation*[12] (BPMN2) [15] developed by the Object Management Group (OMG) is the latest in a number of attempts to create an industry standard language to describe choreography.[13] The new standard is notable in that it is the first to our knowledge to attempt to include rules for realizability, with the following statement:

There are constraints on how Choreography Activities can be sequenced (through Sequence Flow) in a Choreography ... The basic rule of Choreography Activity sequencing is this:

– *The Initiator of a Choreography Activity MUST have been involved (as Initiator or Receiver) in the previous Choreography Activity.*

This captures our asynchronous projectability condition (A2), but is clearly an incomplete treatment of the conditions for realizability.

The BPMN choreography language has no compositional capability of the kind discussed and advocated in this paper.

12 Conclusion

We believe that the development of advanced distributed collaborations will require techniques that enable designers to construct solutions that are transparently correct so that, as far as possible, realizability is guaranteed. The modeling formalisms used to represent choreographies and the resultant behavior contracts must play the key role in this, and the use of compositional modeling appears to be the natural paradigm to express and analyze the inherent parallelism of distributed behavior.

If choreography-based approaches are to succeed they must provide the expressive power to capture real business collaborations, but also allow realizability to be verified using techniques within the competence of mainstream software engineers. We believe that the ideas we discuss here take us significantly nearer achieving this combination and we hope

[11] At least not without including such amounts as extra fields in the messages. But message content is normally fixed by business considerations or standards, and not open to this kind of change.

[12] At the time of writing at the "OMG Adopted Beta Specification" stage.

[13] Previous attempts include WSCI and WS-CDL.

that this work will help inform future directions in the design of choreography modeling languages. In particular, we suggest that *composition capabilities* should be at the heart of such languages.

References

1. Bocchi L, Honda K, Tuosto E, Yoshida N (2009) A theory of design-by-contract for distributed multiparty interactions. Technical Report, University of Leicester, University of London and Imperial College London
2. Ebert J, Engels G (1994) Observable or invocable behaviour—you have to choose. Technical Report 94-38. Department of Computer Science, Leiden University
3. Fu X, Bultan T, Su J (2004) Conversation protocols: a formalism for specification and verification of reactive electronic services. Theor Comput Sci 328(1–2):19–37
4. Fu X, Bultan T, Su J (2004) Realizability of conversation protocols with message contents. In: ICWS '04: Proceedings of the 2004 IEEE international conference on Web services. IEEE Computer Society, Washington, DC, pp 96–103. doi:10.1109/ICWS.2004.92
5. Hoare C (1985) Communicating sequential processes. Prentice-Hall International, Englewood Cliffs, NJ
6. Hoare C (2006) Why ever CSP? Electronic Notes Theor Comput Sci 162:209–215. doi:10.1016/j.entcs.2006.01.031
7. Honda K, Yoshida, N Carbone M, (2008) Multiparty asynchronous session types. SIGPLAN Not 43(1):273–284. doi:10.1145/1328897.1328472
8. Kazhamiakin R, Pistore M (2006) Analysis of realizability conditions for web service choreographies. In: Najm E et al (ed) Proceedings of FORTE 2006, 26th IFIP WG 6.1 international conference, Paris. Lecture Notes in Computer Science, vol 4229. Springer, pp 61–76
9. Lanese I, Guidi C, Montesi F, Zavattaro G (2008) Bridging the gap between interaction- and process-oriented choreographies. In: Proceedings of SEFM'08, 6th IEEE international conferences on software engineering and formal methods. IEEE Computer Society, Washington, DC, pp 323–332
10. McNeile A, Simons N (2006) Protocol modelling: a modelling approach that supports reusable behavioural abstractions. J Softw Syst Model 5(1):91–107 doi:10.1007/s10270-005-0100-7
11. McNeile A, Simons N (2008) Metamaxim Website: ModelScope Tool. http://www.metamaxim.com/
12. Meyer B (2000) Object-oriented software construction. Prentice Hall PTR, Englewood Cliffs, NJ
13. Mendling J, Hafner M, (2005) From inter-organizational workflows to process execution: generating BPEL from WS-CDL. In: Meersman R, Tari Z et al (eds) OTM workshops. Lecture Notes in Computer Science, vol 3762. Springer, pp 506–515
14. Milner R (1980) A calculus of communicating systems. Lecture Notes in Computer Science, vol 92. Springer
15. Object Management Group (2009) Business process model and notation (BPMN): FTF Beta 1 for Version 2.0. OMG Document Number: dtc/2009-08-14. Technical Report, Object Management Group
16. Owicki S, Lamport L (1982) Proving liveness properties of concurrent programs. ACM Trans Program Lang Syst 4(3): 455–495. doi:10.1145/357172.357178